REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 8, 1997	Quarterly Technical Report

4. TITLE AND SUBTITLE
Quarterly Technical Report
Massive Data Analysis Systems

5. FUNDING NUMBERS
F19628-95-C-0194

6. AUTHOR(S)
Richard Frost, Mike Wan, Vibby Gottemukkala
Chaitanya Baru, Reagan Moore, Anant Jhingran
Richard Marciano, Joe Lopez

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
San Diego Supercomputer Center
PO Box 85608
San Diego CA 92186-85608

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ARPA/ITO
Att: Charlene Veney
3701 N. Fairfax Drive
Arlington VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
N/A

11. SUPPLEMENTARY NOTES

DISTRIBUTION STATEMENT A
Approved for public release:
Distribution Unlimited

12a. DISTRIBUTION/AVAILABILITY STATEMENT
ARPA/ITO
ESC/ENS
DTIC
ARPA Technical Library

12b. DISTRIBUTION CODE

19970113 086

13. ABSTRACT *(Maximum 200 words)*

The creation of a Massive Data Analysis System (MDAS) will enable new modes of science through improved data management of scientific data sets. This requires a scalable software infrastructure that can manage petabytes of data, support rapid access of selected data sets, and provide support for subsequent computationally intensive analyses. To accomplish this, object-relational database technology is being integrated with archival storage systems. By supporting transportable methods for manipulating the data, it then becomes possible to analyze selected data sets on remote systems. The MDAS becomes a scheduling system, managing the flow of data and computation across distributed resources. Usage models are needed that simplify the identification, transport and analysis of large collections of data. The system must automate the collection of metadata describing the data set attributes, and handle interactive WEB access, distributed database access, and discipline specific application interfaces. A software infrastructure has been designed which manages user access restrictions, matches application requirements with resource availability, and schedules the data movement and application execution. Development of this software system is proceeding on schedule, with selected applications testing the initial prototypes.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Scientific Data Mining	221
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

Report on Work in Progress: Massive Data Analysis Systems

QUARTERLY TECHNICAL REPORT
October 1996 – December 1996

San Diego Supercomputer Center

Reagan Moore. Principal Investigator.
Chaitanya Baru, Richard Frost, Joe Lopez, Richard Marciano,
Arcot Rajasekar, and Mike Wan

Sponsored by: Advanced Research Projects Agency ITO

ARPA Order No. D007 and D309
Issued by ESC/ENS under Contract F19628–95–C–0194

Distribution:

ARPA Agent (2 copies)
ESC/ENS
Bldg 1704, Rm 114
5 Eglin Street
Hanscom AFB, MA 01731-2116

ARPA/ITO (2 copies)
Attn: Charlene Veney
3701 N. Fairfax Drive
Arlington, VA 22203-1714

ESC/ENK (letter)
Attn: Ms Carole Stephan
Bldg 1704, Rm 119
5 Eglin Stree
Hanscom AFB, MA 01731-2116
(Letter of Transmittal Only)

Defense Technology Information Center (2 copies)
Cameron Station
Alexandria, VA 22034-6145

ARPA/Technical Library (1 copy)
3701 N. Fairfax Drive
Arlington, VA 22203-1714

Source:

Richard Frost
San Diego Supercomputer Center
P.O. Box 85608
San Diego, CA 92186-9784

Contents

List of Figures

List of Tables

ABSTRACT

The creation of a Massive Data Analysis System (MDAS) will enable new modes of science through improved management tools for scientific data sets, computational methods, and computational resources. To provide these capabilities, MDAS researchers are developing a software infrastructure to support data location transparency, access transparency, and conversion transparency in a heterogeneous, distributed systems. This requires a scalable software infrastructure that can manage petabytes of data, support rapid access of selected data sets, and provide support for subsequent computationally intensive analyses. The system must automate the collection of metadata describing data sets, computational methods, resources, and users. Some of the core technologies being used to provide this functionality include object-relational database systems, archival storage systems, parallel I/O, third-party transfers, and method-level authentication. By supporting transportable methods for manipulating the data, it then becomes possible to analyze selected data sets on remote systems. The MDAS becomes an infrastructure to build next-generation operating and scheduling systems which can manage the flow of data and computation across distributed resources.

Chapter 1

Task Objectives

MDAS is an information discovery system that supports queries against metadata stored for users, methods, data sets, and resources. A request for processing of data can be issued based entirely on the objective of the researcher. The information discovery system determines the names of the data set and the methods that need to be applied to the data set, the locations of the data set and method, and the identity of the computer resources that could be used to execute the methods. The result is a program graph that can be given to a scheduler and then to an execution environment. All data set accesses are represented as invocation of methods. This results in a seamless link to information, driven by user needs.

The overall objective of the Massive Data Analysis System is the construction of a scalable system that integrates data management with computation to support analysis and assimilation of arbitrarily large data sets. The initial goals for the project were the development of consensus on (a) application requirements, (b) hardware and software systems that can meet the application requirements, (c) and the application presentation interfaces needed to make the system accessible to researchers. These goals have been met.

Currently, MDAS researchers are concentrating on the implementation of the software design, the use of real applications to test the implementation, the integration of archival storage with database management systems, and the involvement of a commercial software vendor. Integrated data and object handling systems have the potential to be the next major software infrastructure in the evolution of computer systems. If designed correctly, the data handling system can support a uniform user interface to distributed heterogeneous systems needed for global computing. The data handling system acts as a scheduling system that resides above the operating system. Users interact directly with data attributes, relying on the data handling system to locate the data, move the data to appropriate compute resources, and execute applications on the data sets.

Chapter 2

Technical Problems

The primary challenges in the MDAS project are (a) the integration of data management systems with archival storage and (b) end-user solutions for the replacement of the (Unix) file paradigm with a higher-order interface to data, methods, and resources.

At project onset, database management systems (DBMSs) and hierarchical (archive) storage systems (HSSs) were not interoperable. Replacements for the Unix OS file paradigm existed, but only on single-user systems and largely without high-order interfaces to resources. In order to develop concrete extensible solutions with reasonable technology transfer attributes, we have taken a two-pronged approach to these problems. First, we rely heavily on application prototypes to investigate the viability of possible solutions to the primary challenges. Second, we continue to carry out an in-depth research into the design and specification of production-grade software solutions. A number of technical problems have emerged from these efforts which we will address throughout the remainder of the project:

1. Hierarchical storage system (HSSs) have a limited number of I/O channels and physical read/write heads for archival (tape) storage. Therefore, an application with large resource requirements can easily monopolize the entire storage system unless a reasonable resource management policy is implemented in the HSS. This amounts to implementing a queuing system within the HSS to mitigate requests from multiple clients.

2. HSS software interfaces contain general I/O routines such as read, seek, rewind, etc. However, most HSS application client interfaces limit data transactions to file get and put—primarily to keep applications from monopolizing resources. For example, an application controlling a tape drive with seeks and rewinds for 2 hours leads to poor resource utilization for the general user population. This means that when an HSS client wants to read a large file, it must have either (a) enough local memory (RAM or Disk) to read the file in total, or (b) access to an intelligent spooler which can buffer the read and make incremental calls to the HSS for large file blocks. The latter case is most common.

3. The development of an MDAS software infrastructure for general I/O interoperability between local and remote resources requires (a) library interfaces to individual

resources, (b) a high-level interface to hide individual resource semantics from the users, and (c) a high-level interface that supports multiple I/O paradigms across each supported resource. Not all platforms will have suitable library interfaces for all resources; e.g., a desired DBMS, HSS, or HTTP library might be missing. Hence, daemons must be developed to assist application clients on those platforms. Further, the build environment for the MDAS software can become arbitrarily complex without careful design considerations. Rather than "port" the MDAS software to each vendor hardware platform, the MDAS build environment should support compile-time configurable options for drivers to various resources.

Supporting a a high-level interface to hide individual resource semantics from users requires the development of intermediate buffering mechanisms just below the application level. For example, supporting a { read, seek, rewind } paradigm on top of a dataset opened (transparently) on an `ftp` resource will require local buffering and possibly remote re-reads of the file. A related issue is the support of legacy software systems; e.g., the use of Fortran unit numbers for I/O transactions on a DBMS large object, or the transfer of two valid file handles to a third-party data mover for a Unix-style "pipe" transaction.

Tension also exists between items "b" and "c" when a paradigm from one I/O source (e.g., DBMS query) does not match paradigms supported by another (e.g., Unix fifo pipe). Therefore, categories of I/O paradigms need to be identified and then supported by categories of high-level semantics.

4. Among the key resources that will be supported by MDAS are archival storage systems with a database system front-end. The database systems are used to store metadata and to provide access to the archive data sets. However, depending on the particular database system used, the implementation of this interface may be different. In order to insulate MDAS methods from such implementation issues, the MDAS system must provide appropriate mappings from the standard file I/O interface used by the methods to the actual interface supported by each database system front end.

5. Application clients, DBMSs, and HSSs all have response time limitations for I/O and general communication transactions. Coupling these systems requires careful design considerations to avoid request timeouts and blocked (hung) communication requests. For any particular component in the system, it is worthwhile to know in advance that another component is off-line—rather than to wait on an internal (possibly long) connection timeout condition.

6. Application clients, servers, DBMSs, and HSSs all have various authentication mechanisms which can vary among sites. In a distributed environment, it is often desirable to transfer methods and/or datasets from one resource to another for more efficient processing. This capability requires authentication interfaces between coupled systems (e.g., a tightly coupled DBMS and HSS) plus third party authentication mechanisms to permit a server to transfer a client's request to an external (third party) processing system.

7. Object-oriented (OO) software technologies greatly simplify the task of software engineering and hold great promise for software reuse. However, present-day OO compilers do not produce high-performance executables which is of paramount importance to

this project. Hence, MDAS implementations should choose application-efficient languages while providing interfaces to OO language semantics.

8. Applets (as implemented in the Java language) and other interpreted methods extend the OO paradigm to remote resources. However, applets are extremely slow in processing scientific datasets or performing moderate iterative tasks in general. For applets to be truly efficient, "just-in-time" compilation methods are desirable on the target platform.

9. Traditional DBMS and FTP technologies rely on sequential I/O streams for transferring data objects. This approach has been demonstrated to give relatively poor performance unless aggressive caching strategies are developed. The concern is that focusing on just caching strategies will be inappropriate for the more advanced technology that is based on parallel I/O streams.

10. Scientific applications should be able to access data and cache it locally no matter where the data is originally located. This is equivalent to requiring a catalog or expert system with universal resource name (URN) capabilities.

11. Support for parallel I/O streams must be done within the context of emerging standards. This requires tracking the MPI2 IO effort which is examining issues related to message passing within a compute platform and I/O to external peripherals. Interoperability between MPI and non-MPI processes will require specialized software interfaces.

12. The design of appropriate experiments to test the capabilities of the proposed system requires independent testing of individual infrastructure components. This requires dedicating separate portions of the testbed system to HSS support and to database support. The result is that it will be possible to quantify the memory, disk cache, and I/O requirements independently for each system, and then quantify what the integrated system will need in terms of hardware resources to have adequate performance.

Chapter 3

General Methodology

Scientists have an ever-increasing need to store, access, and manipulate unprecedented quantities of data. Modern data manipulation requirements include searches for correlations in large data sets, incorporation of empirical data to improve the predictive capabilities of computational simulations, and mining of existing data sets to derive better input conditions for new calculations. High performance data assimilation environments [8] require new modes of operation to integrate data mining and supercomputing. These large-scale and national-scale problems strain our current infrastructure and motivate the development of massive data analysis systems.

In order to implement a system that can meet these requirements, mutually-scalable technologies are needed in parallel data-handling systems, computational servers, local area networks, and resource scheduling environments. Further, these system complexities need to be presented to users in a set of navigatable hierarchies. This latter criteria insures that a novice can have a high degree of success in using the system, while experts can achieve major advances in analysis and resource efficiencies.

Other efforts in this area have focused exclusively on scalable I/O [11, 3], high-performance computation [7], high-performance communication [5], digital libraries [10], or object-oriented software integration [6]. These are valuable efforts which will provide an experience base for future system integrations. However, in order to be successful, we need to address all aspects of of the "massive data analysis" problem. It is far more effective to design these new-generation systems from a first principles approach, while taking available technology into consideration.

The Massive Data Analysis System (MDAS) provides a software infrastructure, including user-level application program interfaces, metadata catalogs, MDAS *engine* functions and daemons, and MDAS drivers, to enable resource discovery and to identify resources for scheduling computations in a heterogeneous, distributed system. The objective of MDAS is to enable the construction of scalable systems which integrate data management with resource management to support storage, movement, analysis, and display of arbitrarily large data sets. MDAS is being used as one of the software components of the Distributed Object Computation Testbed (DOCT). MDAS provides a seamless link to information; a data management system for heterogeneous data resources.

MDAS defines four types of system entities, viz. Data Set, Resource, Method, and User. Metadata definitions are provided for each type of entity. The system maintains metadata for all "registered" entities and provides the ability to create, update, store, and query this metadata. The metadata is used by the system to perform resource discovery and resource scheduling. Each data set access is represented by an access method which can be scheduled as a part of the overall application. Entities can be registered with MDAS at installation time or by authorized users during the life of the system. Applications can link to libraries which provide routines for accessing the metadata as well as for accessing MDAS entities.

The type of metadata maintained by the system is an extension of the metadata maintained by typical operating systems. The specific goal in MDAS is to employ this metadata to support the storage, handling, and processing of massive data sets. Applications can use the MDAS application program interface library to access registered data sets, resources, and methods. To serve a specific application request, the MDAS engine may perform a variety of actions including: (i) invoking an underlying method or sequence of methods, (ii) allocating a set of resources, (iii) accessing the neccesary data sets, (iv) moving data sets from storage to compute resources, and back, (v) moving data sets among storage resources, e.g. for caching and/or replication, and (vi) plugging in built-in "glue" functions to move data sets between methods.

3.1 Transparency

The objective of the MDAS project is to construct a prototype of a scalable system which integrates data management with computation to support analysis and assimilation of arbitrarily large data sets. The system must provide the ability to store, transport, and process large data sets. By storing metadata and providing the ability to represent, store, query, and process this metadata, the system is able to integrate heterogeneous data management and computational resources in a distributed system. An abstraction of the services available in such a system is provided by supporting various types of *transparencies*.

The wide range of services offered by MDAS are accessible from a set of application program interfaces (API's). These API's provide *location transparency*, thereby enabling applications to access data sets based on their attribute values rather than by their precise location, such as URL or file path. In addition to *location transparency*, MDAS provides a strong abstraction of the distributed system by supporting (i) *Format Transparency* which hides the details of data set formats and structure from the application, (ii) *Method Transparency* which enables applications to specify high-level computational requests that automatically invoke the appropriate method (or sequence of methods) to complete the request, (iii) *Resource transparency* where the system determines the set of resources to use for satisfying a given computational request rather than requiring the application to explicitly state this as an input requirement, and (iv) *User Transparency* which allows MDAS to serve as a broker or surrogate and utilize resources on behalf of users who may not have direct resource authorization.

3.1.1 Location Transparency

A data set can be identified based on intrinsic content or the content of its associated, application-specific metadata, rather than by exact location, such as a URL or file path. This feature provides *location independence* to applications and also allows for optimization of data set accesses. The system can choose the least cost location for accessing a given data set if the data set is cached and/or replicated at multiple sites.

3.1.2 Format Transparency

The system maintains metadata on data sets as well as on function (or *methods*) that operate on data sets. This permits MDAS to distinguish between the form and content of *data sets*. Multiple, existing data sets may have the same semantic content, but may differ from each other in terms of the internal representations. MDAS defines a *change of format* to be invariant on the data set content. An example is a text file which may be converted to tagged HTML format, and further transformed into a Postscript file. By maintaining the neccesary metadata, the system is able to match requests for data sets by identifying the one with the correct format, or by applying conversion methods to convert an existing data set into the desired format.

3.1.3 Method Transparency

For a given computational request, there may be several alternative implementations (i.e., executions of methods on resources) which provide the desired functionality. Given a high-level request, the MDAS library is able to compute and execute an optimal or near-optimal implementation of the request based on user-specified (or default) performance criteria. In some cases MDAS may need to synthesize a sequence of methods to provide the desired functionality. Where possible, metadata is maintained regarding *speed-up* and *scale-up* behavior. These features of the system are referred to as *method transparency*. Data access is viewed as the invocation of a method appropriate for the storage resource being accessed. Method transparency implies that the requestor of the data set does not have to specify the access mechanism.

3.1.4 Resource Transparency

Users can identify specific resources for the execution of a request or simply ask that a request be abstractly executed on a *resource pool*. In the latter case, an optimal (or near optimal) set of resources is identified by MDAS for executing the request according to user-specified (or default) performance criteria. Once again, this is achieved by maintaining appropriate metadata on resources and methods. For resources, the system maintains metadata on histories of load throughput, estimates of queue waiting times, and functions to access current resource load estimates.

3.1.5 User Transparency

In heterogeneous, distributed systems one can expect resources to belong to different security *realms* or *domains*. A single computational request may involve multiple resources belonging to multiple realms. For example, a data set may be moved from a resource at one geographic location to another resource at another location, where it may undergo a conversion, and then moved to a third location where it is processed. All of these resources may operate in different security realms.

If the user submitting the original request has been initially authenticated by MDAS or one of its methods, the MDAS *ticket* mechanism can be used to ensure that subsequent accesses to all the resources is transparent in terms of authentication and security actions. In particular, resources may receive requests directly from MDAS and might not be aware of the identity of the user submitting the original request. Thus, *user transparency* provides an authentication broker service that allows user requests to be processed on resources to which the original user may not have been directly authenticated. Given that data accesses are represented by the access method, user transparency is achieved by authenticating interactions between methods.

Providing user transparency also allows the MDAS system to avoid copying data sets. For example, when a user requests a read-only copy of a data set, MDAS simply increments the user reference counter for the object instead of performing a disk copy. In this manner, the MDAS "User Space" concept replaces the Unix "home directory" paradigm. (MDAS has no notion of files and directories, except in the context of data set storage locations.) A "User Space" is defined by the system entities referenced by a user and is not necessarily a physical space on a resource.

Chapter 4

Technical Results

The MDAS software system is now in the implementation phase. An overview of the MDAS architecture is given in section 4.1. Application scenarios are discussed in section 4.2. A draft tutorial and portions of a user guide for the MDAS Application Program Interface (API) is given in sections 4.3–4.5. Sections 4.6–4.10 comprise the main sections of a draft MDAS API Reference manual. Portions of a draft implementors guide are given in sections 4.11–4.13. Executable versions of MDAS Library routines and miscellaneous tools to assist with the task of Catalog metadata registration are listed in section 4.14. Optional MDAS agents and service brokers are discussed in section 4.15. An early DBMS–Archival Storage integration effort is presented in section 4.16.

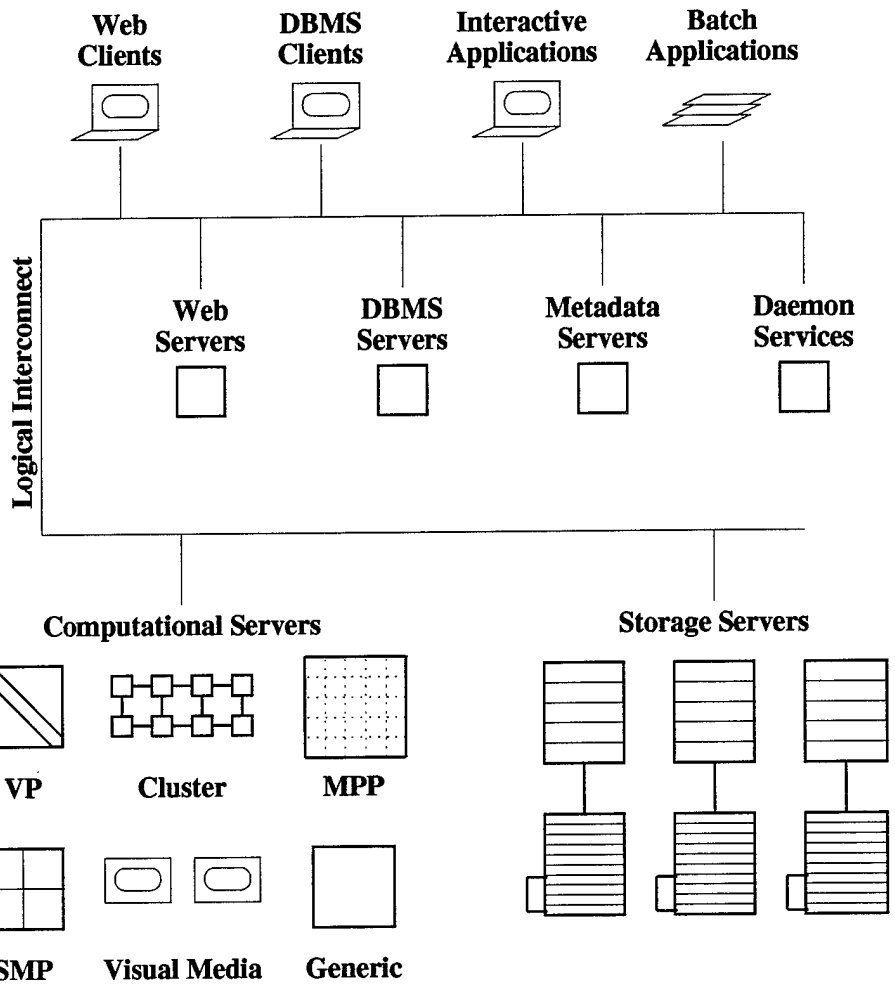

Figure 4.1: Massive data analysis system components.

4.1 MDAS Architecture

To enable transparent use of resources, methods, and data sets, MDAS must generalize disjoint protocols. As such, MDAS is *middleware* that supports a very high-level interface to application layers while maintaining an internal layer of drivers to interface with actual system components. The following subsections identify system components commonly encountered in an MDAS computing environment.

4.1.1 System Components

A generic representation of a computing environment is given in figure 4.1. This diagram is intended to represent system resources and services that users might encounter in a typical computational data analysis environment. The following list describes some of the characteristics of the resources and services available in the computing environment shown

in figure 4.1.

- A "Logical Interconnect" may be an intranet or internet along any network lines. Intra- or internets may exist between computational servers.
- Components represented as single systems may in fact be multiple, distributed, or overlapping.
- Web and DBMS clients are special instances of interactive applications.
- Web and DBMS servers also represent enhanced storage services.
- A Metadata server is a DBMS that stores metadata (both, application-specific metadata as well as MDAS system metadata).
- A Web client is used to browse and download moderate-size data sets or data set segments. A DBMS client is used to query application-specific metadata.
- Batch applications might run on a desktop or a (remote) computational server.
- A daemon service might be something simple like a print spooler or X-Window client—or a complex task scheduling and management system.

4.1.1.1 Users/Clients

MDAS users may access the system from a variety of sources including interactive Web access, direct access via an application program, and access via a DBMS. Users may be *anonymous* or *authenticated*. An authenticated user is one who has previously "registered" with MDAS. MDAS provides mechanisms by which an anonymous user may use the Registration Services of MDAS to become a registered user. Non-registered users are treated as anonymous users and all anonymous users receive default levels of access and service from MDAS.

Along with authentication information, users can be associated with level of service, level of access, and *path histories* of method invocation. The level of service establishes the user's priorities and resource consumption constraints in using the system. Level of access controls the user's accessibility to resources in the system. The user's patterns of use of the MDAS system are cached as *path histories*. They contain information on the nature, frequency, and relationships of accesses to data sets, resources, and methods, to enable future user requests to be served efficiently.

4.1.1.2 Data Sets

Data sets are MDAS entities accessed by methods. These include raw data sets stored on disk or on archival system (or both), data stored under various file systems, and data stored in DBMS's. In addition to standard information such as name, location, size, format, and access control, other information that may be associated with data sets includes partitioning

(especially for parallel data sets), structure, and access history. As mentioned above, a data set access may be in the path history of one or more users. Conversely, a data set itself may be associated with access history information. The path and access history information is important in deciding when a data set needs to be "cached" (e.g. from archival store to disk), when it can be "swapped out", or when it needs to be replicated for improved performance.

4.1.1.3 Methods

Methods in MDAS include programs, macros, and utilities, which operate upon and transform data sets in the system. These may be user-specified or *built-in*. Similar to other entities in the system, when a method is registered, the system stores metadata describing the method, its input/output parameters, its speedup and scaleup characteristics, etc.

4.1.1.4 Servers

In the category of MDAS servers we include computational and storage engines or resources. These entities have a given capacity and therefore the usage/consumption of these resources needs to be monitored for effective scheduling. This includes compute engines (parallel and sequential), visualization engines, interactive systems, and disks. It also includes file servers, DBMS's, and archival storage systems. MDAS maintains metadata on usage of servers.

4.1.1.5 Metadata Servers

TBD.

4.1.2 Authentication

Applications running in a heterogeneous, distributed massive data analysis system may often require connections to resources, methods, and data sets which belong to different security realms or domains. Thus, the system must handle issues related to authentication and security in this environment. A typical operating system provides data set security by maintaining file permissions at the user or group level. In addition, most operating systems use security protocols which rely on passwords and the fact that users first log in to systems using a password prior to starting any other tasks. However, there is a wide variety of service security models that a system may support including, single user model (as in PCs), dedicated network connections (single user networks), password challenge (telnet model), "hard-wired" or pre-set software passwords (software equivalent of dedicated network), friendly host tables (Unix model), and tickets (kerberos, *ssh* model).

The *ticket* model is used within MDAS and, functionally, this model subsumes the other security schemes. The semantics of tickets is based on a point-to-point authorization protocol in which each side is assumed to have a private security key for the other, plus the ability to

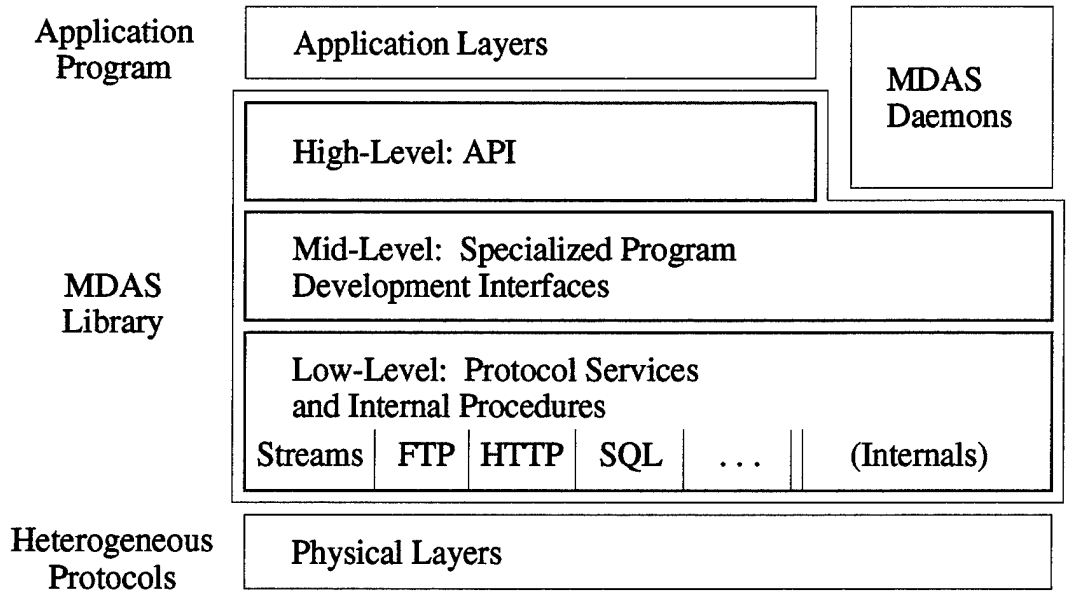

Figure 4.2: MDAS Library placement in an application software hierarchy

generate public "tickets" which can be decoded by the other party's private key for access authorization. Under this model, a passwordless system can be considered to have null tickets. A password challenge system can permit the instantiation of tickets, where the user changes from a login+password paradigm to a ticket+login paradigm. Since the friendly host table model is another instance of pre-set software passwords, it can be replaced by either null or automatically generated tickets. Given the permission to execute a method to access a data set and then to execute an analysis or display method, MDAS supports third-party authentication in which the methods directly exchange tickets to validate the information exchange.

4.1.3 Software Architecture

The system software architecture is composed of a software library plus optional daemons. Figure 4.2 shows a software hierarchy indicating the relationship of MDAS software components. The highest level in the software architecture offers an application program interface (API) to external applications. The middle level consists of the MDAS Library which provides the mechanisms to store, retrieve, update, and act on metadata maintained by MDAS. At the lowest level, MDAS maintains a set of architecture and resource specific drivers which provide the interface between MDAS and the underlying heterogeneous environment.

The library itself may contain generic system functions, e.g. format translation algorithms, but it does not contain application-specific routines. Instead it requires that such methods be "registered" as MDAS methods which will then be invoked as needed. This highlights the role of MDAS as a broker of metadata. The MDAS Library is expected to contain

several value-added methods for standard format translations, spooling operations, etc. The library also provides drivers to interact with MDAS service broker daemons. These daemons provide access to services which may not have been implemented on a particular architecture, e.g. the IBM DB2 database system on SGI/Cray T3E parallel computer. Thus, applications can gain access to such resources even if they are not implemented on the local system.

MDAS entities (including, data sets, methods, resources, and users) are registered in the system either via explicit calls to the appropriate API's, or as a side effect of an MDAS operation. For example, data sets are automatically registered when a new data set is opened for write with an MDAS API call, or when the system creates intermediate result data sets, or when it replicates data sets. In addition, a resource that is new or unknown so far to MDAS is registered when a user explicitly connects to it the first time.

The system metadata is stored in a distributed, replicated MDAS Metadata Catalog. The locations of the MDAS Catalog and its replicas are specified at system installation time on a per-site basis. Users can override or augment the default catalog information at run-time for specific resources.

4.2 Application Scenarios

This section illustrates how MDAS services can be used in a variety of scenarios involving analysis of massive data sets.

4.2.1 Document Text and Image Processing

This scenario illustrates the use of MDAS services for handling text and image data, with specific applicability to the types of applications being considered in the DOCT project.

Patent and trademark related data is stored in archival storage systems using proprietary data formats (e.g. Messenger text files). In order to index these data sets using standard text engines, the original files may have to be converted to standard formats, e.g. SGML, tagged files; cleaned up so that all records in the file are in the same format, e.g. remove leading/trailing blanks; and processed such that relevant metadata is extracted for each file. These converted, "cleaned" files can then be used to create text indexes and populate database systems with the actual document data. The patent files also contain a large number of images stored in the "Yellow Book" format. These files are processed to extract images in standard formats, e.g. tiff images, and store the link between images and patents.

4.2.1.1 Handling Text Files

Queries issued against the patent database may involve reading the processed text indexes associated with the database, as well as reading the original data itself. The system must provide the capability to read this data regardless of whether it is in archival storage or disk, on the local system or a remote system.

Methods used to convert data from archival storage to a form that is suitable for indexing and loading in a database can be registered in MDAS. The system maintains metadata associated with each method. An application can issue the following sequence of requests to MDAS:

1. Connect to an archival storage system containing files for a particular patent, such that, currently, the links from the application host to the the archival system are least congested
2. Connect to a database that currently has sufficient storage space to hold the information for this patent
3. Connect to a text search engine
4. Move all data associated with this patent from the archive to the database and index this data using the text search engine

MDAS metadata is used in responding to the above sequence of requests. For example, the system keeps information on contents and location of data sets and the current state of resources which allows it to identify the archive requested in the first request. Similarly, it stores the location of various resources such as database management systems and text search engines, and their states, which enables it to respond to the second and third requests above. Finally, it keeps tracks of various methods available to it. The last request above prompts the system to search for methods that can carry out the neccesary transformations on the data sets. Thus, it is able to identify the appropriate method(s) to invoke on the specified data sets using the desired set of resources to satisfy the user's requests.

4.2.1.2 Handling Image Data

Similar to the text data, patent files also contain image data which may be stored in archival storage systems and/or disk, and which may be indexed based on image features. The same sequence of requests as above can be issued against image files related to a particular patent. In addition, there may be other types of processing applicable to images. For example, consider the following sequence of requests:

1. Connect to an archival storage containing image data for a given subset of patents
2. Invoke a user defined parallel method which performs image feature extraction and use this to extract all images which satisfy a given condition
3. Store selected images in a database

Metadata associated with the image feature extraction method allows the system to identify the number of nodes needed on a massively parallel computer. Based on the data set sizes, the system may also identify the need for a parallel I/O transfer between the archival storage system and the parallel computer.

4.2.2 Scientific Applications

This section describes a variety of scientific applications which can benefit from the services provided by MDAS.

4.2.2.1 3-D Ocean Simulation Environments

Running very large 3-D ocean simulations (as done by NCCOSC, the Naval Command Control and Ocean Surveillance Center), is both computationally intensive and data intensive. Very large model output files are typically generated for each run. Additional data requirements might include storing information such as model calibration data, observational data for validation, etc.

Creating a historical *log* of model run outputs would be desirable. Unfortunately, current computational simulations are often rerun from scratch in order to reanalyze output data due to storage limitations.

Using the resource metadata of MDAS to identify systems to speed up the simulation itself, as well as the data handling and storage capabilities of MDAS, to store model input and output data sets, would create a unique modeling archive in which post-processing, comparing, and validating of output data would constitute a *modeling audit trail* suitable for use by policy makers and modelers alike.

Model validation tools could be developed in this framework that automate the validation process. Defining classes of statistical tests and associated calibration data sets would be an important step in increasing model result confidence. This could be accomplished in the MDAS framework by registering user-supplied validation methods.

4.2.2.2 Climate Data Assimilation

An important aspect in short-term numerical weather prediction and long-term climate modeling is incorporating observational data into simulation systems (data assimilation). A typical climate data assimilation scenario might involve the following:

1. Observations of the weather system (coming from ground stations, satellites, flying balloons, and other sources) are collected every 6 hours, giving a record of what actually happened.
2. A forecast/simulation model starting from a given initial condition computes successive forecasts every 6 hours for every point of some regular grid (e.g. 2 x 2.5 degrees with 14-22 elevation levels).
3. These two streams of events (what the weather should theoretically be + what was actually observed) are fused together by the data assimilation process, which produces a more optimal data set.

More accurate forecasts can be produced by using the assimilated data set as the initial condition in computing the next forecast, leading to a better forecast sequence.

In operational environments, data assimilation is a very computationally intensive task with real-time requirements specifying that all calculations be completed before the next batch of observation data comes in (6 hours in our example).

The computation and storage requirements involved make this a challenging problem. The application requires access to distributed archival storage systems to be able to handle the storage requirements and proper scheduling of tasks on, possibly, multiple parallel engines. MDAS discovery and resource identification services can be useful here.

4.2.2.3 Digital Sky Survey

Large-area digital sky surveys are a development of astronomical research that can also benefit from the MDAS environment. Recent large-area surveys in optical, infrared, and radio wavelengths will be placed *online* for use by the astronomical community. Collections such as the Digital Palomar Observatory Sky Survey (DPOSS), the 2-Micron All Sky Survey (2-MASS), and the NRAO VLA Sky Survey (NVSS) represent terabytes of data.

Providing access to catalogs and image data with MDAS infrastructure for accessing distributed data archives will allow detailed correlated studies across the entire data set.

4.3 API Tutorial

As a place holder, examples have been included from the reference section.

The following examples assume that **argv** and **argc** are system-defined variables.

4.3.1 Catalog Queries

4.3.2 Fetching Data

```
PROGRAM getk2
     ...
     MDAS_status  status
     MDAS_INFOH   dsinfo, cacheinfo
     ...
     MDAS_INIT(argc, argv, NULL, status)
     ...
     MDAS_INFO_CREATE(MDAS_DATASET, dsinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_NAME, "fin.dat", dsinfo, status)
     MDAS_INFO_CREATE(MDAS_DATASET, cacheinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_STOR_FMTN, "khoros2", cacheinfo, status)
     MDAS_GET(dsinfo, cacheinfo, status)
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM
```

4.3.3 Piping Data Sets

```
PROGRAM pipefun
     ...
     MDAS_status  status
     MDAS_INFOH   funinfo, psinfo
     ...
     MDAS_INIT(argc, argv, NULL, status)
     ...
     MDAS_INFO_CREATE(MDAS_DATASET, funinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_NAME, "fun.dat", funinfo, status)
     ...
     MDAS_INFO_CREATE(MDAS_RESOURCE, psinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_RSRC_TYPE, MDAS_PRINTER, psinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_RSRC_FMTN, "postscript", psinfo, status)
     ...
     MDAS_INFO_PIPE(funinfo, psinfo, status)
```

```
      print "fun.dat printed at:"
      MDAS_INFO_PRINT(psinfo,status)
      ...
      MDAS_FINALIZE(NULL,status)
      ...
END PROGRAM
```

4.3.4 Executing Requests

4.3.5 Computing with User-defined Formats

4.3.6 Connecting to Resources

```
PROGRAM dbconnect
      ...
      MDAS_status  status
      MDAS_INFOH   dbinfo
      MDAS_SERVH   dbh
      ...
      MDAS_INIT(argc, argv, NULL, status)
      ...
      MDAS_INFO_CREATE(MDAS_RESOURCE, dbinfo, status)
      MDAS_INFO_SET_ATTR(MDAS_NAME, "ord.com", dbinfo, status)
      MDAS_INFO_SET_ATTR(MDAS_RSRC_SRVN, "illustra", dbinfo, status)
      MDAS_CONNECT(dbinfo, NULL, NULL, servh, status)
      ...
      MDAS_DISCONNECT(servh, NULL, status)
      ...
      MDAS_FINALIZE(NULL,status)
      ...
END PROGRAM
```

4.3.7 Interacting with MPI

The following example assumes that `argv` and `argc` are system-defined variables. The handle `MPI_COMM_WORLD` is defined by MPI.

```
PROGRAM do_mpi
      ...
      integer ierr
      MDAS_status status
      ...
      MPI_INIT(ierr)
      ...
```

```
      MDAS_INIT(argc, argv, MPI_COMM_WORLD, status)
      ...
      MDAS_FINALIZE(MPI_COMM_WORLD,status)
      ...
      MPI_FINALIZE(ierr)
      ...
END PROGRAM
```

4.4 Mid-Level Tutorial

TBD.

4.5 Run-Time Environment

Intro ...TBD.

Need to discuss how `MDAS_INIT()` is involved with this process.

4.5.1 User and Installation Defined Parameters

TBD.

4.5.2 Default Parameter Locations

TBD.

4.5.3 Command-Line Arguments

TBD.

4.5.4 Environment Variables

TBD.

4.5.5 "Resource" Files

TBD.

4.5.6 "Ticket" Files

TBD.

4.6 Language Bindings

The naming of MDAS library routines and data types varies according to programming language. This avoids ambiguities in mixed language programs.

The MDAS Library is a *functional* rather than an *object-oriented* design. The MDAS Library may be encapsulated in C++ application class libraries but does not directly supply a class library interface.

4.6.1 Library Calls

MDAS Library call names are implemented with the prefix `mdas*_` where * is `F` for Fortran 90, `C` for ANSI C, etc. Thus, the routine `MDAS_INIT()` (section ??) is implemented:

```
mdasF_init(argc, argv, comm, status)      (in Fortran 90)

mdasC_init(argc, argv, comm, status)      (in ANSI C, C++)
```

4.6.2 MDAS Types

The implementation of MDAS data type names (sections 4.7.1,4.8.1,4.11.1) is identical to that of MDAS Library calls. For example, the base types `MDAS_string` and `MDAS_status` are implemented:

```
mdasF_string                              (in Fortran 90)
mdasF_status

mdasC_string                              (in ANSI C, C++)
mdasC_status
```

4.6.3 MDAS Tokens

MDAS Tokens are implemented as parameters in Fortran and pre-processor macros in C and C++. Name conflicts cannot arise. They appear in the language exactly as shown in tables throughout this document.

The actual token values in any language are identical. However, values may change across library versions. Users are advised to always use token names instead of token values for portability.

MDAS Data Type	Fortran 90 Type	ANSI C, C++ Type
MDAS_byte	character	unsigned char
MDAS_character	character	char
MDAS_string	character array	char[] with '\0'
MDAS_logical	logical	integer
MDAS_integer	integer	long
MDAS_real	real	float
MDAS_double	double precision	double
MDAS_complex	complex	double[2]
MDAS_handle	pointer	void*

Table 4.1: MDAS API base data types and their counterparts in standard languages.

4.7 Application Program Interface

MDAS provides a High-Level application program interface (API) for simple interactions with MDAS metadata and services. It is expected that most user needs will be met at this level. Library writers and system software developers may also find the MDAS Mid-Level architecture described in section 4.8 of interest.

This section discusses the MDAS High-Level API in detail. Datatypes exposed to the user at this level are discussed in section 4.7.1. Function prototypes are presented in section 4.7.3.

4.7.1 API Data Types

The MDAS API defines several base data types for interoperability with standard languages types. A list of currently supported types is given in table 4.1. MDAS also defines a set of extended data types discussed in detail below. Attributes of MDAS extended data types are given in table 4.2.

The implementation of these types is language and architecture dependent (see section 4.6). Type conversions between languages and architectures is performed by Mid-Level routines in the MDAS library.

4.7.1.1 Status

Most MDAS Library calls return a status vector `MDAS_status`, which is an integer array of size 4. The use of each element is summarized in table 4.3.

```
status: (IN/OUT)  MDAS_status
```

The procedures `MDAS_STATUS_MSG()`, `MDAS_STATUS_PRINT()`, and `MDAS_STATUS_INFO()` in section 4.7.3.3 provide an interface to MDAS status messages.

MDAS Data Type	Fortran 90 Type	ANSI C, C++ Type
MDAS_status	integer(4)	int[4]
MDAS_time	derived TYPE	struct
MDAS_size	integer(2)	long[2]
MDAS_spectrum	doubleprecision(8)	double[8]
comm	MPI communicator	MPI communicator
MDAS_token	integer	int
MDAS_DATAH	derived TYPE	struct
MDAS_INFOH	derived TYPE	struct
MDAS_SERVH	derived TYPE	struct

Table 4.2: MDAS API extended data types.

status element	Purpose	Values
status(0)	Flag	error: status(0) < 0 success: status(0) = 0 warning: status(0) > 0
status(1)	Total errors and warnings	from 0 to 32
status(2)	Codes	parameterized bit codes
status(3)	Procedure#	from 1 to MDAS_PROC_COUNT

Table 4.3: The status vector. Bit codes are procedure-specific. See procedure definition for list of applicable bit codes.

By definition, status(0) = 0 means success. When status(0) < 0, an error has occurred. Warnings are indicated by status(0) > 0. The value returned in status(1) indicates the total number of errors and warnings which occurred.

Array element status(2) contains up to 32 of status (bit) codes packed into a single integer by logical "or" operations. All status codes have predefined exposed parameter or macro names in the implementation language. (See the definition of each call for a list of applicable status codes.) Bit codes placed in status(2) may be unique to the calling procedure.

The Library call *procedure id#* is returned in status(3). Procedure id#'s range in value from 1 to MDAS_PROC_COUNT.

4.7.1.2 Time

TBD.

MDAS Handle Type	Reference
comm	MPI communicator
MDAS_DATAH	physical contents of a data set
MDAS_INFOH	MDAS_info structure
MDAS_SERVH	service connection

Table 4.4: MDAS handle references.

4.7.1.3 Size

TBD.

4.7.1.4 Comm

Some versions of the MDAS Library are built with MPI [12] for portability on parallel computing platforms. The *MPI communicator* argument comm appears in some MDAS Library routines to extend the MDAS interface to MPI programs. MPI can always be suppressed in MDAS Library routines by passing NULL for comm.

4.7.1.5 Handles

The MDAS Library provides a *handle type* for transparent interface to Info structures, connections, open data sets, and other protocols. A list of MDAS_handle types is given in table 4.4. An MDAS_handle may have an associated handle structure for the purpose of caching Mid-Level library attributes. The handle comm is always address equivalent to an MPI communicator (or NULL). For more detail on MDAS handle structures, see section midlevel-handles.

4.7.2 Info

A central data type in MDAS is the MDAS_info structure, commonly referred to as Info. Its primary use is the capture and specification of metadata from/to the MDAS API. More specifically, Info structures are used to specify information about

- *entities*
- entity *attributes*
- entity *auxilary data*

Info structures are opaque to the calling language and managed by a set of API functions. Implementations of Info structures are language dependent.

4.7.2.1 Info Semantics

The MDAS Info structure provides a means for users to supply un-ordered, incomplete metadata to the MDAS Library for correlation with complete metadata sets stored in an MDAS Catalog. This section discusses the semantic interpretation of user-supplied Info in the context of MDAS Library requests.

Complete metadata specifications are dense with information which is (a) necessary for arbitrary system transactions and (b) typically unknown by end-users. As such, the MDAS Library provides information discovery. A library interface for this purpose must provide a means for

1. users to easily specify what they know
2. users and the library to easily extract what they need to know
3. the library to efficiently store partial and full metadata information
4. the library to translate partial information into Catalog queries

Further, system-level metadata is inherently heirarchical. For example, data sets have attributes of names, summary information, storage information, etc. For example, storage attributes (in a distributed system) of a data set must indicate the number of logical storage replications. Each storage replication may have attribute of being partitioned, in which case each attributes of the segment storage (e.g., location, segmenting method) must be maintained. Historically, users tracked such information in a log book. Part of the MDAS design is to track these details for the user.

The approach to these problems in MDAS is to allow users to specify "what they know" about an entity with a sequential protocol; i.e., a list of known attributes of the entity. Thus, the creation of Info structures is always in the context of an entity. Any additional records added to the structure are then taken to be attributes or auxilary data of the entity. In particular, users should be able to specify very little about an entity let MDAS determine the remaining system level details required to carry out their transaction.

For example, suppose a user knows the name of a particular data set and that it is the member of some group defined in the MDAS Catalog. To create Info about these attributes of the data set, a user program executes a command to

> "create an Info structure of entity type MDAS_DATASET."

To add attributes containing the data set name and group, a user program follows the Info creation instruction with

> "set an Info attribute containing name" and
> "set an Info attribute containing group."

In this manner, the user is provided with a simple, sequential interface for data entry and the library can take care of the details of laying the data out in nested hierarchies of (incomplete) metadata. This meets goal #1.

Goal #2 is to provide users and internal MDAS library procedures with a straightforward method of reading values from an `Info` structure. For example: suppose that a query to an MDAS Catalog has returned a fully populated `Info` structure containing metadata about a particular data set and a user desires to extract metadata attributes of the data set into the memory of a user program.

Three things are required to achieve this functionality: (a) the user presumably knew that MDAS maintains such information, (b) the user must determine what names or function in MDAS can be used to specify the attribute of interest to the MDAS Library, and (c) the library must define a protocol for returning attribute information to the user. In order to obtain goal #1, MDAS adopted the semantics of building `Info` structures by adding attributes to entities. Thus, users will have the expectation of extracting `Info` values in a like manner. To support this expectation, MDAS provides a "seek and read" protocol based on the same attribute names that users provide to input entity attributes. For example, to obtain the common name of a data set from a "Catalog populated" `Info` structure into program variable A, the user would

> "scan the value for attribute `MDAS_NAME` of the `MDAS_DATASET` entity into A."

Goal #3 (efficient in-memory store of metadata) could be achieved by ignoring #1 and #2 and implementing `Info` structures as a C `struct` with no interface to the members other than direct naming of the structure members. This would not only degrade the user interface, but expose the underlying structure to direct user manipulation and place the Library at risk. At the same time, it is prudent to implement `Info` structures as C or F90 structures—but with private members served only to the user by the interface described for #1 and #2. Note that this approach permits long-term evolutions in the `Info` structure design without sacrificing the portability of early implementations. Further, it allows the internal representation of an `Info` structure to be "flattened out" and reduce the number of dereferences required to extract a given value. Thus, the complexity of the internal `Info` structure can increase for the benefit of efficiency without sacrificing the simplicity of the user interface.

To achieve "ease of translation" of data in `Info` format to Catalog queries (goal #4), the MDAS Library incrementally caches SQL-like structures in `Info` based on the entity type and attributes. A convenience routine is then provided to convert partial entity descriptions in `Info` structures to SQL. An alternate approach would be to ask the end user to use SQL directly. It is expected that this would be too great a burden on most users. It also would make Catalog design changes difficult to implement transparently. However, for those users that need a SQL interface MDAS provides a SQL Exec function. A table of entity attributes and their associated tables for the current implementation are given in the MDAS User Guide.

4.7.2.2 Base Entities

4.7.2.2.1 MDAS_DATASET

MDAS_DATASET attribute	Type	Use
(Base)		
MDAS_NAME	MDAS_string	Name of data set. When specified in the info argument of MDAS_INQUIRE(), matches to data sets with this name or alias will be sought. When scanning this attribute from an Info structure that resulted from an inquiry, the "common" name of the data set will be returned. A name may only be registered once. Any further name registrations will produce a new alias.
MDAS_ID	MDAS_integer	Catalog id#. Can be used in inquiries or scanned from inquiry results. Value is never set by user.
MDAS_CD_ID	MDAS_integer	Id# of MDAS Catalog in which this data set can be found. Can be used in inquiries or scanned from inquiry results. MDAS uses this value to differentiate between multiple catalogs referenced during a single run-time. Value is set by library and is only valid during run-time.

MDAS_DATASET attribute	Type	Use
(Version)		
MDAS_VERSION	MDAS_string	MDAS Version # of data set. When the "version" of a data set is updated, it keeps the same names and alias' but obtains a new Catalog id#. Creating a new "version" of a data set is synonymous with re-writing the data set with new or different *content*. If the content is unchanged but in a different format or storage medium, MDAS considers this a *replication*. See MDAS_STOR_FMTN and MDAS_REPLICATES for comparison.
MDAS_VERSIONX	MDAS_logical	Indicates whether MDAS_VERSIONM is an external method registered in the MDAS Catalog or an internal MDAS Library method. If external, MDAS_VERSIONX evaluates to "true".
MDAS_VERSIONM	MDAS_integer	Depending upon the value of MDAS_VERSIONX, MDAS_VERSIONM is either the Catalog id# of an MDAS_METHOD or an MDAS Library token for an internal method. See table ?? for a list of applicable tokens.
MDAS_VERSIONP	MDAS_integer	Catalog id# of previous version (if any) of this data set. This is useful for automatically propagating new alias' to previous versions.
MDAS_VERSIONN	MDAS_integer	Catalog id# of next version (if any) of this data set. This is particularly useful for automatically propagating new alias' to later versions.

MDAS_DATASET attribute	Type	Use
(Alias)		
MDAS_ALIAS	MDAS_string	Alternate names for the data set. On input to MDAS_INQUIRE(), this is functionally equivalent to MDAS_NAME. When scanning this attribute from an Info structure that resulted from an inquiry, an element of the MDAS_ALIASV array will be returned. When preparing an MDAS_DATASET Info structure for registration, adding this attribute will also add an element to the MDAS_ALIASV array.
MDAS_ALIASC	MDAS_integer	# of strings in MDAS_ALIASV.
MDAS_ALIASV	MDAS_string array	Array of alias strings. To read this attribute, first read MDAS_ALIASC and allocate an array of that size. Alternately, incrementally read MDAS_ALIAS. To set this attribute, first set MDAS_ALIASC or set MDAS_ALIAS incrementally.

MDAS_DATASET attribute	Type	Use
(Documentation)		
MDAS_ABSTRACT	MDAS_integer	Catalog id# of Abstract. Every abstract is an independent entity. To add an abstract, first register the abstract and obtain its Catalog id#.
MDAS_DOC	MDAS_integer	Catalog id# of Documentation. Every set of documentation is an independent entity. To add documentation, first register the doc's and obtain their Catalog id#.

MDAS_DATASET attribute	Type	Use
(SD)		
MDAS_SD_TABLE	MDAS_integer	Catalog id# of relational table which contains this data set (and possibly others) as a LOB in a column cell. To add this attribute, first register the table as an MDAS_DATASET and then obtain the Catalog id#. **Note:** To register a data set with SD attributes, all SD attributes must be specified.
MDAS_SD_KEYC	MDAS_integer	# of relational columns in MDAS_SD_TABLE applicable to MDAS inquiries. Identifies the # of elements in MDAS_SD_COLS, MDAS_SD_KEY_TYP, and MDAS_SD_KEY.
MDAS_SD_COLS	MDAS_string array	The names of columns in MDAS_SD_TABLE applicable to MDAS inquiries. To read this attribute, first read MDAS_SD_KEYC and allocate an array of that size. To set this attribute, first set MDAS_SD_KEYC.
MDAS_SD_KEY_TYP	MDAS_integer	The (MDAS) data types of the columns named in MDAS_SD_COLS.
MDAS_SD_KEYS	MDAS_handle	The values in the columns of MDAS_SD_TABLE which uniquely identify this data set. To read this attribute, first read MDAS_SD_KEYC and MDAS_SD_KEY_TYP, then allocate an appropriate structure or array. To set this attribute, first set MDAS_SD_KEYC, MDAS_SD_COLS, and MDAS_SD_KEYS.
MDAS_SD_LOB_COL	MDAS_string	Name of SD column cell containing LOB for this data set.

MDAS_DATASET attribute	Type	Use
(Logical Group)		
MDAS_STOR_GRPN	MDAS_string	Name of some MDAS_GROUP in which this data set has membership. When specified in the info argument of MDAS_INQUIRE(), matches to data sets with this group will be sought. When scanning this attribute from an Info structure that resulted from an inquiry, an element of the MDAS_STOR_GRPNV array will be returned. When preparing an MDAS_DATASET Info structure for registration, adding this attribute will also add an element to the MDAS_STOR_GRPNV array.
MDAS_STOR_GRPI	MDAS_integer	Catalog id# of some MDAS_GROUP in which this data set has membership. When specified in the info argument of MDAS_INQUIRE(), matches to data sets with this group will be sought. When scanning this attribute from an Info structure that resulted from an inquiry, an element of the MDAS_STOR_GRPIV array will be returned. When preparing an MDAS_DATASET Info structure for registration, adding this attribute will also add an element to the MDAS_STOR_GRPIV array.
MDAS_STOR_GRPNC	MDAS_integer	# of MDAS_GROUP names in MDAS_STOR_GRPNV array.
MDAS_STOR_GRPNV	MDAS_string array	Array of MDAS_GROUP names in which this data set has membership. MDAS_STOR_GRPNV is the union of group memberships across any replicates of the data set. A group membership need not span all replicates. To read this attribute, first read MDAS_STOR_GRPNC and allocate an array of that size. Alternately, incrementally read MDAS_STOR_GRPN. To set this attribute, first set MDAS_STOR_GRPNC or set MDAS_STOR_GRPN incrementally.
MDAS_STOR_GRPIC	MDAS_integer	# of MDAS_GROUP Catalog id#s in MDAS_STOR_GRPIV array.
MDAS_STOR_GRPIV	MDAS_integer array	Array of MDAS_GROUP Catalog id#'s in which this data set has membership. MDAS_STOR_GRPIV is the union of group memberships across any replicates of the data set. A group membership need not span all replicates. To read this attribute, first read MDAS_STOR_GRPIC and allocate an array of that size. Alternately, incrementally read MDAS_STOR_GRPI. To set this attribute, first set MDAS_STOR_GRPIC or set MDAS_STOR_GRPI incrementally.

MDAS_DATASET attribute	Type	Use
(Logical Domain)		
MDAS_STOR_DMNN	MDAS_string	Name of some MDAS_DOMAIN in which this data set has membership. When specified in the info argument of MDAS_INQUIRE(), matches to data sets with this domain will be sought. When scanning this attribute from an Info structure that resulted from an inquiry, an element of the MDAS_STOR_DMNNV array will be returned. When preparing an MDAS_DATASET Info structure for registration, adding this attribute will also add an element to the MDAS_STOR_DMNNV array.
MDAS_STOR_DMNI	MDAS_integer	Catalog id# of some MDAS_DOMAIN in which this data set has membership. When specified in the info argument of MDAS_INQUIRE(), matches to data sets with this domain will be sought. When scanning this attribute from an Info structure that resulted from an inquiry, an element of the MDAS_STOR_DMNIV array will be returned. When preparing an MDAS_DATASET Info structure for registration, adding this attribute will also add an element to the MDAS_STOR_DMNIV array.
MDAS_STOR_DMNNC	MDAS_integer	# of MDAS_DOMAIN names in MDAS_STOR_DMNNV array.
MDAS_STOR_DMNNV	MDAS_string array	Array of MDAS_DOMAIN names in which this data set has membership. MDAS_STOR_DMNNV is the union of domain memberships across any replicates of the data set. A domain membership need not span all replicates. See MDAS_REPL_DMNNA. To read this attribute, first read MDAS_STOR_DMNNC and allocate an array of that size. Alternately, incrementally read MDAS_STOR_DMNN. To set this attribute, first set MDAS_STOR_DMNNC or set MDAS_STOR_DMNN incrementally. MDAS_STOR_DMNNV contains all names in MDAS_REPL_DMNNA.
MDAS_STOR_DMNIC	MDAS_integer	# of MDAS_DOMAIN Catalog id#s in MDAS_STOR_DMNIV array.
MDAS_STOR_DMNIV	MDAS_integer array	Array of MDAS_DOMAIN Catalog id#'s in which this data set has membership. MDAS_STOR_DMNIV is the union of domain memberships across any replicates of the data set. A domain membership need not span all replicates. See MDAS_REPL_DMNIA. To read this attribute, first read MDAS_STOR_DMNIC and allocate an array of that size. Alternately, incrementally read MDAS_STOR_DMNI. To set this attribute, first set MDAS_STOR_DMNIC or set MDAS_STOR_DMNI incrementally. MDAS_STOR_DMNIV contains all names in MDAS_REPL_DMNIA.

MDAS_DATASET attribute	Type	Use
(Input Lineage)		
MDAS_STOR_IN	MDAS_integer	Catalog id# of an MDAS_DATASET source used in the creation of this data set (if any). Applies only to the origination of data set, not replications or segments. When inquiring about data sets, MDAS_STOR_IN may be used to match source "lineage". When scanning MDAS_STOR_IN from Info returned by MDAS_INQUIRE(), the value is some element of MDAS_STOR_INV. When adding attributes to an Info structure for Catalog registration, set MDAS_STOR_INC and use MDAS_STOR_INV.
MDAS_STOR_INC	MDAS_integer	# of items listed in MDAS_STOR_INV. When inquiring about data sets, MDAS_STOR_INC may be used to match the count of data sets in source "lineage". When scanning MDAS_STOR_INC from data set Info returned by MDAS_INQUIRE(), the value is the actual count for a particular data set. When adding attributes to an Info structure for Catalog registration, set MDAS_STOR_INC before making incremental adds of MDAS_STOR_IN or adding the MDAS_STOR_INV vector.
MDAS_STOR_INV	MDAS_integer array	Array of Catalog an MDAS_DATASET id#'s that specify the sources used to create this data set. Applies only to the origination of data set, not replications or segments. The ordering of inputs specified in MDAS_STOR_INV must match the ordering of inputs defined in the Catalog metadata for MDAS_STOR_GEN.

MDAS_DATASET attribute	Type	Use
(Consequential Lineage)		
MDAS_STOR_OUT	MDAS_integer	Catalog id# of an entity produced from this data set. When inquiring about data sets. MDAS_STOR_OUT may be used to match output "lineage". When scanning MDAS_STOR_OUT from Info returned by MDAS_INQUIRE(), the value returned is some element of MDAS_STOR_OUTV. When adding attributes to an Info structure for Catalog registration, incremental addition of the MDAS_STOR_OUT attribute will add elements to MDAS_STOR_OUTV and may increment MDAS_STOR_OUTC if necessary. To avoid ambiguous behavior, scan or set MDAS_STOR_OUTC and use MDAS_STOR_OUTV.
MDAS_STOR_OUTC	MDAS_integer	# of items listed in MDAS_STOR_OUTV.
MDAS_STOR_OUTV	MDAS_integer array	Array of Catalog id#'s of entities created from this data set. Applies only to the logical data set, not replicates or segments.

MDAS_DATASET attribute	Type	Use
(Generation Lineage)		
MDAS_STOR_GEN	MDAS_integer	Catalog id# of the MDAS_METHOD which generated this data set. When inquiring about data sets, MDAS_STOR_GEN may be used to match method "lineage". When scanning MDAS_STOR_GEN from data set Info returned by MDAS_INQUIRE(), the value returned is the method which generated the data set. The ordering of inputs specified in MDAS_STOR_INV must match the ordering of inputs defined in the Catalog metadata for MDAS_STOR_GEN. Applies only to the origination of data set, not replications or segments.
MDAS_STOR_GNP	MDAS_string	The parameter list (if any) used with MDAS_STOR_GEN to generate this data set. When inquiring about data sets, MDAS_STOR_GNP may be used to match method "lineage". When scanning MDAS_STOR_GNP from data set Info returned by MDAS_INQUIRE(), the value returned is the parameter list used to generate the data set. Applies only to the origination of data set, not replications or segments. The format of MDAS_STOR_GNP must match the parameter format required for MDAS_STOR_GEN.
MDAS_STOR_GNR	MDAS_integer	Catalog id# of the MDAS_RESOURCE on which MDAS_STOR_GEN executed to create the data set. When inquiring about data sets, MDAS_STOR_GNR may be used to match method "lineage". Applies only to the origination of data set, not replications or segments.
MDAS_STOR_GNU	MDAS_integer	Catalog id# of the MDAS_USER which executed MDAS_STOR_GEN to create the data set. When inquiring about data sets, MDAS_STOR_GNU may be used to match method "lineage". Applies only to the origination of data set, not replications or segments.

MDAS_DATASET attribute	Type	Use
(Re-Generation Policy)		
MDAS_STOR_PLCY	MDAS_integer	Update policy for this data set. When inquiring about data sets, MDAS_STOR_PLCY may be used to match the policy of an arbitrary data set. When scanning MDAS_STOR_PLCY from data set Info returned by MDAS_INQUIRE(), the value represents the data set update policy. When adding attributes to an Info structure for Catalog registration, only one MDAS_STOR_PLCY may be set.

MDAS_DATASET attribute	Type	Use
(Trigger)		
MDAS_STOR_TRGM	MDAS_integer	Catalog MDAS_METHOD id# of some trigger for this data set. These methods are executed when updates are made to this data set and the corresponding trigger policy evaluates to "true". When inquiring about data sets, MDAS_STOR_TRGM may be used to match data set triggers. When scanning MDAS_STOR_TRGM from data set Info returned by MDAS_INQUIRE(), the value returned is some element of MDAS_STOR_TRGMV. When adding attributes to an Info structure for Catalog registration, incremental addition of the MDAS_STOR_TRGM attribute will add elements to MDAS_STOR_TRGMV and may increment MDAS_STOR_TRGC if necessary. To avoid ambiguous behavior, scan or set MDAS_STOR_TRGC and use MDAS_STOR_TRGMV.
MDAS_STOR_TRGP	MDAS_integer	Catalog MDAS_POLICY id# of some trigger policy for this data set. Triggers are executed when updates are made to this data set and the corresponding trigger policy evaluates to "true". When inquiring about data sets, MDAS_STOR_TRGP may be used to match data set policies. When scanning MDAS_STOR_TRGP from data set Info returned by MDAS_INQUIRE(), the value returned is some element of MDAS_STOR_TRGPV. When adding attributes to an Info structure for Catalog registration, incremental addition of the MDAS_STOR_TRGP attribute will add elements to MDAS_STOR_TRGPV and may increment MDAS_STOR_TRGC if necessary. To avoid ambiguous behavior, scan or set MDAS_STOR_TRGC and use MDAS_STOR_TRGPV.
MDAS_STOR_TRGC	MDAS_integer	# of triggers and trigger policies listed in MDAS_STOR_TRGMV and MDAS_STOR_TRGPV.
MDAS_STOR_TRGMV	MDAS_integer array	Array of method Catalog id#'s. These methods are executed when updates are made to this data set and the corresponding policy in MDAS_STOR_TRGPV evaluates to "true".
MDAS_STOR_TRGPV	MDAS_integer array	Array of policy Catalog id#'s for triggers in MDAS_STOR_TRGMV.

MDAS_DATASET attribute	Type	Use
(Storage Date)		
MDAS_STOR_DATE	MDAS_time	Creation date of data set. It is typically scanned or specified with MDAS_INFO_SCAN_ATTR(info, MDAS_STOR_DATE, odate, status) and MDAS_INFO_SET_ATTR(MDAS_STOR_DATE, odate, info, status). If the data set has been replicated, MDAS_STOR_DATE is also the creation date of the "original" replicate. See MDAS_REPL_DATE and MDAS_REPL_DATEV.

MDAS_DATASET attribute	Type	Use
(Storage Permanence)		
MDAS_STOR_PERM	MDAS_double	Probability of not being purged (permanence). MDAS_STOR_PERM applies to all instances of the data set. If the data set has been replicated, MDAS_STOR_PERM represents the cumulative probability across all replications (all elements of MDAS_REPL_PERMV). See MDAS_REPLICATES.
MDAS_STOR_PURG	MDAS_logical	True if data set has been purged. MDAS_STOR_PURG applies to all instances of the data set. If the data set has been replicated, MDAS_STOR_PURG represents the conjunctive truth across all replications (all elements of MDAS_REPL_PURGV). See MDAS_REPLICATES.

MDAS_DATASET attribute	Type	Use
(Storage Size)		
MDAS_STOR_SIZE	MDAS_size	The size of the data set, in bytes. When inquiring about data sets, MDAS_STOR_SIZE may be used to match the size of an arbitrary data set or data set replicate (see MDAS_REPLICATES below). When scanning MDAS_STOR_SIZE from Info returned by MDAS_INQUIRE(), the value may represent either (a) the size of an unreplicated data set or (b) the size of some replicate of the data set (an element of MDAS_REPL_SIZEV). When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_SIZE with MDAS_INFO_SET_RATTR().

MDAS_DATASET attribute	Type	Use
(Storage Format)		
MDAS_STOR_FMTN	MDAS_string	The name (or alias) of the data set format. When inquiring about data sets, MDAS_STOR_FMTN may be used to match the format of an arbitrary data set or data set replicate (see MDAS_REPLICATES below). When scanning MDAS_STOR_FMTN from Info returned by MDAS_INQUIRE(), the name may represent either (a) the format of an unreplicated data set or (b) the format of some replicate of the data set (an element of MDAS_REPL_FMTNV). When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_FMTN with MDAS_INFO_SET_RATTR() and MDAS_INFO_SCAN_RATTR().
MDAS_STOR_FMTI	MDAS_integer	The Catalog id# of the data set format. When inquiring about data sets, MDAS_STOR_FMTI may be used to match the format of an arbitrary data set or data set replicate (see MDAS_REPLICATES below). When scanning MDAS_STOR_FMTI from Info returned by MDAS_INQUIRE(), the value may represent either (a) the format of an unreplicated data set or (b) the format of some replicate of the data set (an element of MDAS_REPL_FMTIV). When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_FMTI with MDAS_INFO_SET_RATTR() and MDAS_INFO_SCAN_RATTR().

MDAS_DATASET attribute	Type	Use
(Storage Resource)		
MDAS_STOR_RSCN	MDAS_string	The name (or alias) of a storage resource for the data set. When inquiring about data sets, MDAS_STOR_RSCN may be used to match the name of an arbitrary storage resource. When scanning MDAS_STOR_RSCN from Info returned by MDAS_INQUIRE(), the name may represent either (a) the storage resource of an unreplicated data set or (b) the storage resource of some replicate of the data set (an element of MDAS_REPL_RSCNV). When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_RSCN with MDAS_INFO_SET_RATTR().
MDAS_STOR_RSCI	MDAS_integer	The Catalog id# of a storage resource for the data set format. When inquiring about data sets, MDAS_STOR_RSCI may be used to match the id# of an arbitrary storage resource. When scanning MDAS_STOR_RSCI from Info returned by MDAS_INQUIRE(), the value may represent either (a) the id# of a storage resource for an unreplicated data set or (b) the id# of a storage resource for some replicate of the data set (an element of MDAS_REPL_RSCIV). When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_RSCI with MDAS_INFO_SET_RATTR().

MDAS_DATASET attribute	Type	Use
(Storage Server)		
MDAS_STOR_SRVN	MDAS_string	The name (or alias) of a software server for the data set. This might be the O/S for the storage resource, or a specialized service provider. When inquiring about data sets, MDAS_STOR_SRVN may be used to match the name of an arbitrary software server. When scanning MDAS_STOR_SRVN from Info returned by MDAS_INQUIRE(), the name may represent either (a) the software server for an unreplicated data set or (b) the software server for some replicate of the data set (an element of MDAS_REPL_SRVNV). When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_SRVN with MDAS_INFO_SET_RATTR().
MDAS_STOR_SRVI	MDAS_integer	The Catalog id# of a software server for the data set format. This might be the O/S for the storage resource, or a specialized service provider. When inquiring about data sets, MDAS_STOR_SRVI may be used to match the id# of an arbitrary software server. When scanning MDAS_STOR_SRVI from Info returned by MDAS_INQUIRE(), the value may represent either (a) the id# of a software server for an unreplicated data set or (b) the id# of a software server for some replicate of the data set (an element of MDAS_REPL_SRVIV). When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_SRVI with MDAS_INFO_SET_RATTR().

MDAS_DATASET attribute	Type	Use
(Storage Path and Name) MDAS_STOR_DIR	MDAS_string	The storage "directory" of the data set on the given resource (MDAS_STOR_RSCN or MDAS_STOR_RSCI) and server (MDAS_STOR_SRVN or MDAS_STOR_SRVI). MDAS distinguishes between data set storage *directories* and storage *names*. In the simple Unix or MS DOS case, the full file path is the concatenation of MDAS_STOR_NAM to MDAS_STOR_DIR. When inquiring about data sets, MDAS_STOR_DIR may be used to match the directory of an arbitrary data set or data set replicate (see MDAS_REPLICATES below). When scanning MDAS_STOR_DIR from Info returned by MDAS_INQUIRE(), the value may represent either (a) the directory of an unreplicated data set or (b) the directory of some replicate of the data set (an element of MDAS_REPL_DIRV). When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_DIR with MDAS_INFO_SET_RATTR().
MDAS_STOR_NAM	MDAS_string	The O/S or server name of the data set at the given directory (MDAS_STOR_DIR) on the given resource (MDAS_STOR_RSCN or MDAS_STOR_RSCI) and server (MDAS_STOR_SRVN or MDAS_STOR_SRVI). MDAS distinguishes between data set storage *directories* and storage *names*. In the simple Unix or MS DOS case, the full file path is the concatenation of MDAS_STOR_NAM to MDAS_STOR_DIR. When inquiring about data sets, MDAS_STOR_NAM may be used to match the O/S or server name of an arbitrary data set or data set replicate (see MDAS_REPLICATES below). When scanning MDAS_STOR_NAM from Info returned by MDAS_INQUIRE(), the value may represent either (a) the "storage name" for an unreplicated data set or (b) the "storage name" of some replicate of the data set (an element of MDAS_REPL_NAM). When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_NAM with MDAS_INFO_SET_RATTR().

MDAS_DATASET attribute	Type	Use
(Data Set Owner) MDAS_STOR_OWN	MDAS_integer	Catalog id# of logical owner of the data set or a specific replicate. When inquiring about data sets, MDAS_STOR_OWN may be used to match the owner of an arbitrary data set or data set replicate (see MDAS_REPLICATES below). Scanning MDAS_STOR_OWN from Info returned by MDAS_INQUIRE() with MDAS_INFO_SCAN_ATTR(info, MDAS_STOR_OWN, owner, status) the logical owner of the data set. Use MDAS_REPL_OWN with MDAS_INFO_SET_RATTR() or MDAS_INFO_SCAN_RATTR() to reference the owner of a particular replicate.

MDAS_DATASET attribute	Type	Use
(Data Set SpecHist)		
MDAS_STOR_HSA	MDAS_integer	Catalog id# of some action listed in MDAS_STOR_HSAV for a logical data set. When inquiring about data sets, MDAS_STOR_HSA may be used to match an action tracked for a particular data set. To scan or set MDAS_STOR_HSA Info, use MDAS_STOR_HSC and MDAS_STOR_HSAV.
MDAS_STOR_HST	MDAS_time	Time stamp for some spectral history of an action for a logical data set. When inquiring about data sets, MDAS_STOR_HST may be used to match time stamps of spectral history compilations for actions on a particular data set. To scan or set MDAS_STOR_HST Info, use MDAS_STOR_HSC and MDAS_STOR_HSTV.
MDAS_STOR_HSS	MDAS_spectrum	Spectral histories of an action for a logical data set. When inquiring about data sets, MDAS_STOR_HSS may be used to match spectral history compilations for arbitrary actions on a particular data set. To scan or set MDAS_STOR_HSS Info, use MDAS_STOR_HSC and MDAS_STOR_HSSV.
MDAS_STOR_HSC	MDAS_integer	# of actions listed in MDAS_STOR_HSAV, MDAS_STOR_HSTV, and MDAS_STOR_HSSV. When inquiring about data sets, MDAS_STOR_HSC may be used to match the count of actions tracked to date for a particular data set. When scanning MDAS_STOR_HSC from data set Info returned by MDAS_INQUIRE(), the value is the actual count for a particular data set. When adding attributes to an Info structure for Catalog registration, set MDAS_STOR_HSC before making incremental adds of MDAS_STOR_HSAV or MDAS_STOR_HSSV.
MDAS_STOR_HSAV	MDAS_integer array	Array of Catalog id#'s of actions logical data set. When inquiring about data sets, MDAS_STOR_HSAV may be used to match a set of actions tracked for a particular data set. To scan MDAS_STOR_HSAV Info, first scan MDAS_STOR_HSC and allocate an array of that size, then use MDAS_INFO_SCAN_ATTR(info, MDAS_STOR_HSAV, userarray, status).
MDAS_STOR_HSTV	MDAS_time array	Time stamps of spectral history compilations for a logical data set. When inquiring about data sets, MDAS_STOR_HSTV may be used to match the time stamp array for spectral history compilations on a particular data set. To scan MDAS_STOR_HSTV Info, first scan MDAS_STOR_HSC and allocate an array of that size, then use MDAS_INFO_SCAN_ATTR(info, MDAS_STOR_HSTV, userarray, status).
MDAS_STOR_HSSV	MDAS_spectrum array	Spectral histories for a logical data set. When inquiring about data sets, MDAS_STOR_HSSV may be used to match entire spectral history sets a particular data set. To scan MDAS_STOR_HSSV Info, first scan MDAS_STOR_HSC and allocate an array of that size, then use MDAS_INFO_SCAN_ATTR(info, MDAS_STOR_HSSV, userarray, status).

MDAS_DATASET attribute	Type	Use
(Data Set Perf) MDAS_STOR_PERF	TBD	TBD.

MDAS_DATASET attribute	Type	Use
(Data Set Lock) MDAS_STOR_RLCK	MDAS_logical	Read access lock flag for logical data set (all replicates and segments). "True" means read access to data set is locked.
MDAS_STOR_RLCKS	MDAS_time	Start time for read access lock flag for logical data set (all replicates and segments).
MDAS_STOR_RLCKE	MDAS_time	End time for read access lock flag for logical data set (all replicates and segments).
MDAS_STOR_RLCKD	MDAS_integer	Domain Catalog id# which is allowed to set read locks on logical data set.
MDAS_STOR_WLCK	MDAS_logical	Write access lock flag for logical data set (all replicates and segments). "True" means write access to data set is locked.
MDAS_STOR_WLCKS	MDAS_time	Start time for write access lock flag for logical data set (all replicates and segments).
MDAS_STOR_WLCKE	MDAS_time	End time for write access lock flag for logical data set (all replicates and segments).
MDAS_STOR_WLCKD	MDAS_integer	Domain Catalog id# which is allowed to set read locks on logical data set.

MDAS_DATASET attribute	Type	Use
(Data Set Security) MDAS_STOR_AUTK	MDAS_string	Public authorization key for logical data set (all replicates and segments).
MDAS_STOR_AUTM	MDAS_integer	Method Catalog id# for public authorization key.
MDAS_STOR_RACCK	MDAS_string	Public read-access key for logical data set (all replicates and segments).
MDAS_STOR_RACCM	MDAS_integer	Method Catalog id# for public read-access key.
MDAS_STOR_WACCK	MDAS_string	Public write-access key for logical data set (all replicates and segments).
MDAS_STOR_WACCM	MDAS_integer	Method Catalog id# for public write-access key.
MDAS_STOR_CRYK	MDAS_string	Public encryption key for logical data set (all replicates and segments).
MDAS_STOR_CRYM	MDAS_integer	Method Catalog id# for public encryption key.

MDAS_DATASET attribute	Type	Use
(Replicates) MDAS_REPLICATES	MDAS_integer	# of data set replications. Replications are always equivalent in content, but not necessarily in format or storage media. By default, an unreplicated data set is replicate #1. All replications have the same MDAS_ID, names, and alias'. Replicates can be generated by either (a) a format or storage translation on an instance of the data set, or (b) re-execution of the method that generated the original instance of the data set but using different target format and/or storage parameters. In either case, the storage "Input Lineage" (MDAS_STOR_INV) and "Generation Lineage" (MDAS_STOR_GEN/GNP/GRC/GNU) will be the same but "Replication Lineage" attributes will vary. Replicate attributes may be accessed with either MDAS_INFO_{SET/SCAN}_ATTR() or MDAS_INFO_{SET/SCAN}_RATTR(). See individual attribute for details. For case 'a', methods which perform format translations but otherwise do not alter the content of entities have MDAS_INVARIENT set "true". Data sets can also be segmented (partitioned). See MDAS_REPL_SEGC.

MDAS_DATASET attribute	Type	Use
(Replicate Group)		
MDAS_REPL_GRPN	MDAS_string	Name of some MDAS_GROUP in which this replicate has membership. When specified in the info argument of MDAS_INQUIRE(), matches to data set replicates with this group will be sought; i.e., inquiries with MDAS_REPL_GRPN are identical to inquiries with MDAS_STOR_GRPN. Use MDAS_REPL_GRPNV to scan or register all the groups for a particular data set replicate.
MDAS_REPL_GRPNC	MDAS_integer array	MDAS_REPL_GRPNC$_i$ is the # of MDAS_GROUP names for replicate #i.
MDAS_REPL_GRPNV	MDAS_string array	Array of MDAS_GROUP names for some data set replicate. When specified in the info argument of MDAS_INQUIRE(), matches to replicates with memberships in these particular groups will be sought. The same group memberships need not span all replicates. To scan this attribute, use MDAS_INFO_SCAN_RATTR(info, i, MDAS_REPL_GRPNV, grpnames, status). To set this attribute, first use MDAS_INFO_SET_RATTR(i, MDAS_REPL_GRPNC, grpcount, info, status) then MDAS_INFO_SET_RATTR(i, MDAS_REPL_GRPNV, grpnames, info, status).
MDAS_REPL_GRPI	MDAS_integer	Catalog id# of some MDAS_GROUP in which this replicate has membership. When specified in the info argument of MDAS_INQUIRE(), matches to data set replicates with this group will be sought; i.e., inquiries with MDAS_REPL_GRPI are identical to inquiries with MDAS_STOR_GRPI. Use MDAS_REPL_GRPIV to scan or register all the groups for a particular data set replicate.
MDAS_REPL_GRPIC	MDAS_integer array	MDAS_REPL_GRPIC$_i$ is the # of MDAS_GROUP Catalog id#'s for replicate #i.
MDAS_REPL_GRPIV	MDAS_string array	Array of MDAS_GROUP Catalog id#'s for some data set replicate. When specified in the info argument of MDAS_INQUIRE(), matches to replicates with memberships in these particular groups will be sought. The same group memberships need not span all replicates. To scan this attribute, use MDAS_INFO_SCAN_RATTR(info, i, MDAS_REPL_GRPIV, grpids, status). To set this attribute, first use MDAS_INFO_SET_RATTR(i, MDAS_REPL_GRPIC, grpcount, info, status) then MDAS_INFO_SET_RATTR(i, MDAS_REPL_GRPIV, grpids, info, status).

MDAS_DATASET attribute	Type	Use
(Replicate Domain)		
MDAS_REPL_DMNN	MDAS_string	Name of some MDAS_DOMAIN in which this replicate has membership. When specified in the info argument of MDAS_INQUIRE(), matches to data set replicates with this domain will be sought; i.e., inquiries with MDAS_REPL_DMNN are identical to inquiries with MDAS_STOR_DMNN. Use MDAS_REPL_DMNNV to scan or register all the domains for a particular data set replicate.
MDAS_REPL_DMNNC	MDAS_integer array	MDAS_REPL_DMNNC$_i$ is the # of MDAS_DOMAIN names for replicate #i.
MDAS_REPL_DMNNV	MDAS_string array	Array of MDAS_DOMAIN names for some data set replicate. When specified in the info argument of MDAS_INQUIRE(), matches to replicates with memberships in these particular domains will be sought. The same domain memberships need not span all replicates. To scan this attribute, use MDAS_INFO_SCAN_RATTR(info, i, MDAS_REPL_DMNNV, dmnnames, status). To set this attribute, first use MDAS_INFO_SET_RATTR(i, MDAS_REPL_DMNNC, dmncount, info, status) then MDAS_INFO_SET_RATTR(i, MDAS_REPL_DMNNV, dmnnames, info, status).
MDAS_REPL_DMNI	MDAS_integer	Catalog id# of some MDAS_DOMAIN in which this replicate has membership. When specified in the info argument of MDAS_INQUIRE(), matches to data set replicates with this domain will be sought; i.e., inquiries with MDAS_REPL_DMNI are identical to inquiries with MDAS_STOR_DMNI. Use MDAS_REPL_DMNIV to scan or register all the domains for a particular data set replicate.
MDAS_REPL_DMNIC	MDAS_integer array	MDAS_REPL_DMNIC$_i$ is the # of MDAS_DOMAIN Catalog id#'s for replicate #i.
MDAS_REPL_DMNIV	MDAS_string array	Array of MDAS_DOMAIN Catalog id#'s for some data set replicate. When specified in the info argument of MDAS_INQUIRE(), matches to replicates with memberships in these particular domains will be sought. The same domain memberships need not span all replicates. To scan this attribute, use MDAS_INFO_SCAN_RATTR(info, i, MDAS_REPL_DMNIV, dmnids, status). To set this attribute, first use MDAS_INFO_SET_RATTR(i, MDAS_REPL_DMNIC, dmncount, info, status) then MDAS_INFO_SET_RATTR(i, MDAS_REPL_DMNIV, dmnids, info, status).

MDAS_DATASET attribute	Type	Use
(Repl. Input Lineage) MDAS_REPL_IN	MDAS_integer	Catalog id# of an MDAS_DATASET source used in the creation of this data set **replicate**. Applies only to the replication of data set, not segments or origination. When inquiring about data sets, MDAS_REPL_IN may be used to match replication "lineage". When scanning MDAS_REPL_IN from Info returned by MDAS_INQUIRE(), the value is some data set used in creating the replication. To obtain all inputs used to create a replicate, use MDAS_REPL_INV. When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_INV.
MDAS_REPL_INC	MDAS_integer	# of items listed in MDAS_REPL_INV. When inquiring about data sets, MDAS_REPL_INC may be used to match the count of data sets in replication "lineage". When scanning MDAS_REPL_INC from data set Info returned by MDAS_INQUIRE(), the value is the actual count of inputs to a replication method used to create a particular data set **replicate**. When adding attributes to an Info structure for Catalog registration, set MDAS_REPL_INC before making incremental adds of MDAS_REPL_IN or adding an MDAS_REPL_INV vector.
MDAS_REPL_INV	MDAS_integer array	Array of Catalog id#'s used as inputs to a replication method that created some data set **replicate**. Applies only to the replication of a data set, not segments or origination. When inquiring about data sets, MDAS_REPL_INC and MDAS_REPL_INV may be used to match replication "lineage". When scanning MDAS_REPL_INV from Info returned by MDAS_INQUIRE(), the value is some vector of input data set id#'s used to create one of the data set replications. When adding attributes to an Info structure for Catalog registration, first use MDAS_INFO_SET_RATTR(*i*, MDAS_REPL_INC, incount, info, status) and then use MDAS_INFO_SET_RATTR(*i*, MDAS_REPL_INV, rlinvect, info, status).

MDAS_DATASET attribute	Type	Use
(Repl. Conseq. Lineage) MDAS_REPL_OUT	MDAS_integer	Catalog id# of an MDAS_DATASET data set **replicate** produced from another data set replicate. Applies only to the replication of data set, not segments or origination. When inquiring about data sets, MDAS_REPL_OUT may be used to match replication "consequential lineage". When scanning MDAS_REPL_OUT from Info returned by MDAS_INQUIRE(), the value is the Catalog id# of some data replicate set. To add this value for Catalog registration, use MDAS_INFO_SET_RATTR(*i*, MDAS_REPL_OUT, rlout, info, status). To obtain all replicates produced from another replicate, use MDAS_REPL_OUTV.
MDAS_REPL_OUTVC	MDAS_integer	# of items listed in MDAS_REPL_OUTV. When inquiring about data sets, MDAS_REPL_OUTVC may be used to match the count of data sets in replication "consequential lineage". When scanning MDAS_REPL_OUTVC from data set Info returned by MDAS_INQUIRE(), the value is the actual count of replicates created from a particular data set **replicate**. When adding attributes to an Info structure for Catalog registration, set MDAS_REPL_OUTVC before making incremental adds of MDAS_REPL_OUT or adding an MDAS_REPL_OUTV vector.
MDAS_REPL_OUTV	MDAS_integer array	Array of Catalog id#'s used as inputs to a replication method that created some data set **replicate**. Applies only to the replication of a data set, not segments or origination. When inquiring about data sets, MDAS_REPL_OUTVC and MDAS_REPL_OUTV may be used to match replication "consequential lineage". When scanning MDAS_REPL_OUTV from Info returned by MDAS_INQUIRE(), use MDAS_INFO_SET_RATTR(*i*, MDAS_REPL_OUTVC, outcount, info, status) and then MDAS_INFO_SCAN_RATTR(*i*, MDAS_REPL_OUTV, rloutvect, info, status). When adding attributes to an Info structure for Catalog registration, first use MDAS_INFO_SET_RATTR(*i*, MDAS_REPL_OUTVC, outcount, info, status) and then use MDAS_INFO_SET_RATTR(*i*, MDAS_REPL_OUTV, rloutvect, info, status).

MDAS_DATASET attribute	Type	Use
(Repl. Gen. Lineage) MDAS_REPL_GEN	MDAS_integer	Catalog id# of an MDAS_METHOD which generated this data set **replicate**. Applies only to the replication of a data set, not segments or origination. When inquiring about data set replicates, MDAS_REPL_GEN may be used to match "replication method lineage". When scanning MDAS_REPL_GEN from data set Info returned by MDAS_INQUIRE(), use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_GEN, rmethod, info, status) to find the method used to produce replicate i. Likewise, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_GEN, rmethod, info, status) to specify the method used to produce replicate i. The ordering of inputs specified in MDAS_REPL_INV must match the ordering of inputs defined in the Catalog metadata for MDAS_REPL_GEN.
MDAS_REPL_GNP	MDAS_string	The parameter list (if any) used with MDAS_REPL_GEN to generate this data set **replicate**. Applies only to the replication of a data set, not segments or origination. When inquiring about data set replicates, MDAS_REPL_GNP may be used to match method "replication method lineage". When scanning MDAS_REPL_GNP from data set Info returned by MDAS_INQUIRE(), use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_GNP, rparams, info, status) to find the method parameters used to produce replicate i. Likewise, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_GNP, rparams, info, status) to specify the parameters used to produce replicate i. The format of MDAS_REPL_GNP must match the parameter format required for MDAS_REPL_GEN.
MDAS_REPL_GNR	MDAS_integer	Catalog id# of the MDAS_RESOURCE on which MDAS_REPL_GEN executed to create the data set **replicate**. Applies only to the replication of a data set, not segments or origination. When inquiring about data set replicates, MDAS_REPL_GNR may be used to match method "replication method lineage". When scanning MDAS_REPL_GNR from data set Info returned by MDAS_INQUIRE(), use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_GNR, replrsrc, info, status) to find the resource where replicate i was produced.
MDAS_REPL_GNU	MDAS_integer	Catalog id# of the MDAS_USER which executed MDAS_REPL_GEN to create the data set **replicate**. Applies only to the replication of a data set, not segments or origination. When inquiring about data set replicates, MDAS_REPL_GNU may be used to match method "replication method lineage". When scanning MDAS_REPL_GNU from data set Info returned by MDAS_INQUIRE(), use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_GNU, repluser, info, status) to find the user that produced replicate i.

MDAS_DATASET attribute	Type	Use
(Replication Trigger) MDAS_REPL_TRGM	MDAS_integer	Catalog MDAS_METHOD id# of some trigger for a data set **replicate**. These replication methods are executed when updates are made to this data set and the corresponding trigger policy evaluates to "true". When inquiring about data sets, MDAS_REPL_TRGM may be used to match data set replication triggers. When scanning MDAS_REPL_TRGM from data set Info returned by MDAS_INQUIRE(), the value returned is some element of MDAS_REPL_TRGMV. When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_TRGC and MDAS_REPL_TRGMV to avoid ambiguous behavior.
MDAS_REPL_TRGP	MDAS_integer	Catalog MDAS_POLICY id# of some trigger policy for a data set **replicate**. Replication triggers are executed when updates are made to this data set replicate and the corresponding trigger policy evaluates to "true". When inquiring about data set replicates, MDAS_REPL_TRGP may be used to match data set replication policies. When scanning MDAS_REPL_TRGP from data set Info returned by MDAS_INQUIRE(), the value returned is some element of MDAS_REPL_TRGPV. When adding attributes to an Info structure for Catalog registration, use MDAS_REPL_TRGC and MDAS_REPL_TRGPV to avoid ambiguous behavior.
MDAS_REPL_TRGC	MDAS_integer	# of triggers and trigger policies listed in MDAS_REPL_TRGMV and MDAS_REPL_TRGPV for some replicate. When scanning MDAS_REPL_TRGC from Info returned by MDAS_INQUIRE(), use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_TRGC, repltrgc, info, status) to find the number of replication triggers for replicate i. Likewise, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_TRGC, repltrgc, info, status) to specify the replication trigger count for replicate i.
MDAS_REPL_TRGMV	MDAS_integer array	Array of method Catalog id#'s. These methods are executed when updates are made to this data set and the corresponding policy in MDAS_REPL_TRGPV evaluates to "true". To scan the entire array for replicate i, first obtain MDAS_REPL_TRGC and allocate an array of that size. Scan values into the array with MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_TRGMV, repltrgmv, info, status).
MDAS_REPL_TRGPV	MDAS_integer array	Array of policy Catalog id#'s for triggers in MDAS_REPL_TRGMV. To scan the entire array for replicate i, first obtain MDAS_REPL_TRGC and allocate an array of that size. Scan values into the array with MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_TRGPV, repltrgpv, info, status).

MDAS_DATASET attribute	Type	Use
(Repl. Re-Gen. Policy) MDAS_REPL_PLCY	MDAS_integer	Update policy for a data set **replicate**. When inquiring about data set replicates, MDAS_REPL_PLCY may be used to match "replication regeneration policies". Applies only to the replication of a data set, not segments or origination. When scanning MDAS_REPL_PLCY from Info returned by MDAS_INQUIRE(), use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_PLCY, replplcy, info, status) to find the policy that produced replicate i. Likewise, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_PLCY, replplcy, info, status) to specify the policy for regeneration of replicate i.

MDAS_DATASET attribute	Type	Use
(Replicate Dates) MDAS_REPL_DATE	MDAS_time	The creation time for some replicate. When inquiring about data sets, MDAS_REPL_DATE may be used to match the creation date of an arbitrary data set replicate. When scanning MDAS_REPL_DATE from Info returned by MDAS_INQUIRE(), the value represents the creation date of some replicate of the data set (an element of MDAS_REPL_DATEV). To obtain the creation time for a specific replicate i, use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_DATE, repldate, info, status). When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_DATE, repldate, info, status).
MDAS_REPL_DATEV	MDAS_time array	MDAS_REPL_DATEV$_i$ is the creation time for replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_DATEV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_DATEV, userarray, info, status).

MDAS_DATASET attribute	Type	Use
(Replicate Permanence) MDAS_REPL_PERM	MDAS_double	Probability of some replicate not being purged (permanence). When inquiring about data sets, MDAS_REPL_PERM may be used to match the permanence of an arbitrary data set replicate. When scanning MDAS_REPL_PERM from Info returned by MDAS_INQUIRE(), use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_PERM, replperm, info, status) to obtain the permanence value for replicate i. When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_PERM, replperm, info, status). See also MDAS_REPL_PERMV.
MDAS_REPL_PERMV	MDAS_double array	MDAS_REPL_PERMV$_i$ is the permanence value for replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_PERMV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_PERMV, userarray, info, status).
MDAS_REPL_PURG	MDAS_logical	True if replicate has been purged. When inquiring about data sets, MDAS_REPL_PURG may be used to match the purge status of an arbitrary data set replicate. When scanning MDAS_REPL_PURG from Info returned by MDAS_INQUIRE(), use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_PURG, replpurg, info, status) to obtain the purge truth value for replicate i. When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_PURG, replpurg, info, status).
MDAS_REPL_PURGV	MDAS_logical array	MDAS_REPL_PURGV$_i$ is true if replicate i has been purged. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_PURGV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_PURGV, userarray, info, status).

MDAS_DATASET attribute	Type	Use
(Replicate Sizes)		
MDAS_REPL_SIZE	MDAS_size	The size of a data set **replicate**. When inquiring about data sets, MDAS_REPL_SIZE may be used to match the size of an arbitrary data set replicate. When scanning MDAS_REPL_SIZE from Info returned by MDAS_INQUIRE(), the value represents the size of some replicate of the data set (an element of MDAS_REPL_SIZEV). To obtain the creation time for a specific replicate i, use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_SIZE, replsize, info, status). When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_SIZE, replsize, info, status).
MDAS_REPL_SIZEV	MDAS_size array	MDAS_REPL_SIZEV$_i$ is the creation time for replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_SIZEV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_SIZEV, userarray, info, status).

MDAS_DATASET attribute	Type	Use
(Replicate Format) MDAS_REPL_FMTH Note: a data set replicate with heterogeneous format segments will not have any data in the format attributes below. See MDAS_REPL_SEGC.	MDAS_logical	True when a data set replicate has heterogeneous format segments. To scan this attribute, use MDAS_INFO_SCAN_RATTR(i, info, MDAS_REPL_FMTH, hflag, status). See MDAS_REPL_SEGC.
MDAS_REPL_FMTN	MDAS_string	The name (or alias) of the format of some data set replicate. When inquiring about data sets, MDAS_REPL_FMTN may be used to match the format of an arbitrary data set replicate. To obtain the format name of a specific replicate i, use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_FMTN, replfmtn, info, status). When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_FMTN, replfmtn, info, status). See MDAS_REPL_FMTNV.
MDAS_REPL_FMTNV	MDAS_string array	MDAS_REPL_FMTNV$_i$ is the format name (or alias) of replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_FMTNV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_FMTNV, userarray, info, status).
MDAS_REPL_FMTI	MDAS_integer	The Catalog id# of the format of some data set replicate. When inquiring about data sets, MDAS_REPL_FMTI may be used to match the format of an arbitrary data set replicate. To obtain the format id# of a specific replicate i, use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_FMTI, replfmti, info, status). When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_FMTI, replfmti, info, status). See MDAS_REPL_FMTIV.
MDAS_REPL_FMTIV	MDAS_integer array	MDAS_REPL_FMTIV$_i$ is the format Catalog id# of replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_FMTIV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_FMTIV, userarray, info, status).

MDAS_DATASET attribute	Type	Use
(Repl. Storage Resource) MDAS_REPL_RSCH Note: a data set replicate with distributed storage segments will not have any data in the resource attributes below. See MDAS_REPL_SEGC.	MDAS_logical	True when a data set replicate has heterogeneous format segments. To scan this attribute, use MDAS_INFO_SCAN_RATTR(i, info, MDAS_REPL_RSCH, hflag, status). See MDAS_REPL_SEGC.
MDAS_REPL_RSCN	MDAS_string	The name (or alias) of a storage resource for some data set replicate. When inquiring about data set replicates, MDAS_REPL_RSCN may be used to match the name of a storage resource for an arbitrary data set replicate. To obtain the storage resource name of a specific replicate i, use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_RSCN, replrscn, info, status). When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_RSCN, replrscn, info, status). See MDAS_REPL_RSCNV.
MDAS_REPL_RSCNV	MDAS_string array	MDAS_REPL_RSCNV$_i$ is the storage resource name (or alias) of replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_RSCNV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_RSCNV, userarray, info, status).
MDAS_REPL_RSCI	MDAS_integer	The Catalog id# of a storage resource for some data set replicate. When inquiring about data set replicates, MDAS_REPL_RSCI may be used to match the id# of a storage resource for an arbitrary data set replicate. To obtain the storage resource id# of a specific replicate i, use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_RSCI, replrsci, info, status). When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_RSCI, replrsci, info, status). See MDAS_REPL_RSCIV.
MDAS_REPL_RSCIV	MDAS_integer array	MDAS_REPL_RSCIV$_i$ is the storage resource Catalog id# of replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_RSCIV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_RSCIV, userarray, info, status).

MDAS_DATASET attribute	Type	Use
(Repl. Storage Server) MDAS_REPL_SRVH Note: a data set replicate with heterogeneous storage service segments will not have any data in the server attributes below. See MDAS_REPL_SEGC.	MDAS_logical	True when a data set replicate has heterogeneous format segments. To scan this attribute, use MDAS_INFO_SCAN_RATTR(i, info, MDAS_REPL_SRVH, hflag, status). See MDAS_REPL_SEGC.
MDAS_REPL_SRVN	MDAS_string	The name (or alias) of a storage resource server for some data set replicate. This might be the O/S for the storage resource, or a specialized service provider. When inquiring about data set replicates, MDAS_REPL_SRVN may be used to match the name of a storage resource server for an arbitrary data set replicate. To obtain the storage resource server name of a specific replicate i, use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_SRVN, replsrvn, info, status). When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_SRVN, replsrvn, info, status). See MDAS_REPL_SRVNV.
MDAS_REPL_SRVNV	MDAS_string array	MDAS_REPL_SRVNV$_i$ is the storage resource server name (or alias) of replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_SRVNV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_SRVNV, userarray, info, status).
MDAS_REPL_SRVI	MDAS_integer	The Catalog id# of a storage resource server for some data set replicate. This might be the O/S for the storage resource, or a specialized service provider. When inquiring about data set replicates, MDAS_REPL_SRVI may be used to match the id# of a storage resource server for an arbitrary data set replicate. To obtain the storage resource server id# of a specific replicate i, use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_SRVI, replsrvi, info, status). When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_SRVI, replsrvi, info, status). See MDAS_REPL_SRVIV.
MDAS_REPL_SRVIV	MDAS_integer array	MDAS_REPL_SRVIV$_i$ is the storage resource server Catalog id# of replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_SRVIV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_SRVIV, userarray, info, status).

MDAS_DATASET attribute	Type	Use
(Repl. Dir. and Name) Note: a data set replicate with heterogeneous storage segments will not have any data in the directory and name attributes below. See MDAS_REPL_SEGC.		
MDAS_REPL_DIRN	MDAS_string	The storage "directory" for some data set replicate. MDAS distinguishes between data set storage *directories* and storage *names*. In the simple Unix or MS DOS case, the full file path is the concatenation of MDAS_REPL_NAM to MDAS_REPL_DIR. When inquiring about data set replicates, MDAS_REPL_DIRN may be used to match the storage directory for an arbitrary data set replicate. To obtain the storage directory of a specific replicate i, use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_DIRN, repldirn, info, status). When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_DIRN, repldirn, info, status). See MDAS_REPL_DIRNV.
MDAS_REPL_DIRNV	MDAS_string array	MDAS_REPL_DIRNV$_i$ is the storage directory of replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_DIRNV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_DIRNV, userarray, info, status).
MDAS_REPL_NAMN	MDAS_string	The O/S or server name for some data set replicate. MDAS distinguishes between data set storage *directories* and storage *names*. In the simple Unix or MS DOS case, the full file path is the concatenation of MDAS_REPL_NAM to MDAS_REPL_DIR. When inquiring about data set replicates, MDAS_REPL_NAMN may be used to match the O/S or server name for an arbitrary data set replicate. To obtain the O/S or server name of a specific replicate i, use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_NAMN, replnamn, info, status). When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_NAMN, replnamn, info, status). See MDAS_REPL_NAMNV.
MDAS_REPL_NAMNV	MDAS_string array	MDAS_REPL_NAMNV$_i$ is the O/S or server name of replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_NAMNV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_NAMNV, userarray, info, status).

MDAS_DATASET attribute	Type	Use
(Replicate Owner)		
MDAS_REPL_OWN	MDAS_string	An MDAS_USER Catalog id# for the owner of some data set replicate. When inquiring about data set replicates, MDAS_REPL_OWN may be used to match the owner's Catalog id# for an arbitrary data set replicate. To obtain the owner id# of a specific replicate i, use MDAS_INFO_SCAN_RATTR(i, MDAS_REPL_OWN, replown, info, status). When adding attributes to an Info structure for Catalog registration, use MDAS_INFO_SET_RATTR(i, MDAS_REPL_OWN, replown, info, status). See MDAS_REPL_OWNV.
MDAS_REPL_OWNV	MDAS_string array	MDAS_REPL_OWNV$_i$ is the storage directory of replicate i. This array can be scanned and set with MDAS_INFO_SCAN_ATTR(info, MDAS_REPL_OWNV, userarray, status) and MDAS_INFO_SET_ATTR(MDAS_REPL_OWNV, userarray, info, status).

MDAS_DATASET attribute	Type	Use
(Replicate SpecHist)		
MDAS_REPL_HSA	MDAS_integer	Catalog id# of some action listed in MDAS_REPL_HSAV for a data set replicate. When inquiring about data sets, MDAS_REPL_HSA may be used to match an action tracked for a data set replicate. To scan or set MDAS_REPL_HSA Info, use MDAS_REPL_HSC and MDAS_REPL_HSAV.
MDAS_REPL_HST	MDAS_time	Time stamp for some spectral history of an action for a data set replicate. When inquiring about data sets, MDAS_REPL_HST may be used to match time stamps of spectral history compilations for actions on a data set replicate. To scan or set MDAS_REPL_HST Info, use MDAS_REPL_HSC and MDAS_REPL_HSTV.
MDAS_REPL_HSS	MDAS_spectrum	Spectral histories of an action for a data set replicate. When inquiring about data sets, MDAS_REPL_HSS may be used to match spectral history compilations for arbitrary actions on a data set replicate. To scan or set MDAS_REPL_HSS Info, use MDAS_REPL_HSC and MDAS_REPL_HSSV.
MDAS_REPL_HSC	MDAS_integer	# of actions listed in MDAS_REPL_HSAV, MDAS_REPL_HSTV, and MDAS_REPL_HSSV. When inquiring about data sets, MDAS_REPL_HSC may be used to match the count of actions tracked to date for a data set replicate. When scanning MDAS_REPL_HSC from data set Info returned by MDAS_INQUIRE(), the value is the actual count for a data set replicate. When adding attributes to an Info structure for Catalog registration, set MDAS_REPL_HSC before making incremental adds of MDAS_REPL_HSAV or MDAS_REPL_HSSV.
MDAS_REPL_HSAV	MDAS_integer array	Array of Catalog id#'s of actions data set replicate. When inquiring about data sets, MDAS_REPL_HSAV may be used to match a set of actions tracked for a data set replicate. To scan MDAS_REPL_HSAV Info, first scan MDAS_REPL_HSC and allocate an array of that size, then use MDAS_INFO_SCAN_RATTR(*i*, info, MDAS_REPL_HSAV, userarray, status).
MDAS_REPL_HSTV	MDAS_time array	Time stamps of spectral history compilations for a data set replicate. When inquiring about data sets, MDAS_REPL_HSTV may be used to match the time stamp array for spectral history compilations on a data set replicate. To scan MDAS_REPL_HSTV Info, first scan MDAS_REPL_HSC and allocate an array of that size, then use MDAS_INFO_SCAN_RATTR(*i*, info, MDAS_REPL_HSTV, userarray, status).
MDAS_REPL_HSSV	MDAS_spectrum array	Spectral histories for a data set replicate. When inquiring about data sets, MDAS_REPL_HSSV may be used to match entire spectral history sets a data set replicate. To scan MDAS_REPL_HSSV Info, first scan MDAS_REPL_HSC and allocate an array of that size, then use MDAS_INFO_SCAN_RATTR(*i*, info, MDAS_REPL_HSSV, userarray, status).

MDAS_DATASET attribute	Type	Use
(Replicate Lock)		
MDAS_REPL_RLCK	MDAS_logical	Read access lock flag for data set replicate (all replicates and segments). "True" means read access to data set is locked.
MDAS_REPL_RLCKS	MDAS_time	Start time for read access lock flag for data set replicate.
MDAS_REPL_RLCKE	MDAS_time	End time for read access lock flag for data set replicate.
MDAS_REPL_RLCKD	MDAS_integer	Domain Catalog id# which is allowed to set read locks on data set replicate.
MDAS_REPL_WLCK	MDAS_logical	Write access lock flag for data set replicate. "True" means write access to data set is locked.
MDAS_REPL_WLCKS	MDAS_time	Start time for write access lock flag for data set replicate.
MDAS_REPL_WLCKE	MDAS_time	End time for write access lock flag for data set replicate.
MDAS_REPL_WLCKD	MDAS_integer	Domain Catalog id# which is allowed to set read locks on data set replicate.

MDAS_DATASET attribute	Type	Use
(Replicate Security)		
MDAS_REPL_AUTK	MDAS_string	Public authorization key for data set replicate.
MDAS_REPL_AUTM	MDAS_integer	Method Catalog id# for public authorization key.
MDAS_REPL_RACCK	MDAS_string	Public read-access key for data set replicate.
MDAS_REPL_RACCM	MDAS_integer	Method Catalog id# for public read-access key.
MDAS_REPL_WACCK	MDAS_string	Public write-access key for data set replicate.
MDAS_REPL_WACCM	MDAS_integer	Method Catalog id# for public write-access key.
MDAS_REPL_CRYK	MDAS_string	Public encryption key for data set replicate.
MDAS_REPL_CRYM	MDAS_integer	Method Catalog id# for public encryption key.

MDAS_DATASET attribute	Type	Use
(Replicate Perf)		
MDAS_REPL_PERF	TBD	TBD.

MDAS_DATASET attribute	Type	Use
(Segment Storage)		
MDAS_REPL_SEGC	MDAS_integer	# of segments in individual data set replication. Any replication may be partitioned. Segments are registered as independent data sets. By default, all replications are composed of 1 segment. Use MDAS_INFO_SCAN_RATTR(i, info, MDAS_REPL_SEGC, replsegc, status) to scan this value for replicate i. Updates made to individual segments effect the parent entity according to the trigger and lineage polices of the parent and child data sets.
MDAS_REPL_SEGM	MDAS_integer	Catalog id# of method used to segment some data set replicate. Use MDAS_INFO_SCAN_RATTR(i, info, MDAS_REPL_SEGM, replmeth, status) to scan this value for replicate i.
MDAS_REPL_SEGRV	MDAS_integer array	Array of segment ranks for Catalog id#'s listed in MDAS_REPL_SEGIV. These ranks provide the linkage between MDAS_REPL_SEGIV and MDAS_REPL_SEGM. Use MDAS_INFO_SCAN_RATTR(i, info, MDAS_REPL_SEGRV, replsegranks, status) to scan this value for replicate i.
MDAS_REPL_SEGIV	MDAS_integer array	Array of data set Catalog id#'s composing the segments of some data set replicate. Use MDAS_INFO_SCAN_RATTR(i, info, MDAS_REPL_SEGIV, replsegids, status) to scan this value for replicate i.

4.7.2.2.2 MDAS_METHOD

Methods are ultimately stored in a computing environment and therefore share many attributes with data sets. Here, all attributes of methods are listed for completeness. Descriptions are only given for those attributes unique to methods.

A method *replicate* is a storage instance of an executable code. Hence, method replicates have read, write, and executable attributes. Replicates of the same method instantiated for different operating systems may have different input, output, and execution parameter constraints.

A method *segment* is a physical segment of an executable code or library. Object files composing a library are an example of segments of an executable library registered as a method.

Methods may be coupled together to form *flows*. A flow is a compound method with specified linkage between its input/output and the inputs/outputs of the member methods, plus specific internal linkages between the inputs and outputs of the members.

Unique Method Attributes

MDAS_METHOD attribute	Type	Use
(Content Variance)		
MDAS_METH_INVAR	MDAS_logical	Invariant method →method changes format, not content.
MDAS_METH_IVRTC	MDAS_integer	Invertible method →input is reproducible from output. MDAS_METH_IVRTC is the count of registered methods that can perform this inversion.
MDAS_METH_IVRTM	MDAS_integer array	If MDAS_METH_IVRT is greater than 0, then MDAS_METH_IVRTM$_m$ is the Catalog id# of a method which can perform output-to-input inversion.
MDAS_METH_IVRTI	MDAS_integer array	For each method m in MDAS_METH_IVRTM, MDAS_METH_IVRTI$_{mk}$ maps the kth output of the invertible method to a given input of MDAS_METH_IVRTM$_m$. MDAS_METH_IVRTI has dimension MDAS_METH_IVRTC by MDAS_METH_OUTC.
MDAS_METH_IVRTO	MDAS_integer array	For each method m in MDAS_METH_IVRTM, MDAS_METH_IVRTI$_{mk}$ designates which output of MDAS_METH_IVRTM$_m$ will reproduce the kth input of the invertible method. MDAS_METH_IVRTI has dimension MDAS_METH_IVRTC by MDAS_METH_INC.

MDAS_METHOD attribute	Type	Use
(Server)		
MDAS_SERV_METH	MDAS_logical	True if method is a server. See MDAS_SERVER.

MDAS_METHOD attribute	Type	Use
(Method Lock)		Additional locking attributes for methods.
MDAS_METH_ELCK	MDAS_logical	Execute access lock flag for logical method (all replicates and segments). "True" means execute access to method is locked.
MDAS_METH_ELCKS	MDAS_time	Start time for execute access lock flag for logical method (all replicates and segments).
MDAS_METH_ELCKE	MDAS_time	End time for execute access lock flag for logical method (all replicates and segments).
MDAS_METH_ELCKD	MDAS_integer	Domain Catalog id# which is allowed to set execute locks on logical method.

MDAS_METHOD attribute	Type	Use
(Method Lock)		Additional locking attributes for methods.
MDAS_REPL_ELCK	MDAS_logical	Execute access lock flag for a replicate of the method. "True" means execute access to method is locked.
MDAS_REPL_ELCKS	MDAS_time	Start time for execute access lock flag for a replicate of the method.
MDAS_REPL_ELCKE	MDAS_time	End time for execute access lock flag for a replicate of the method.
MDAS_REPL_ELCKD	MDAS_integer	Domain Catalog id# which is allowed to set execute locks on a replicate of the method.

MDAS_METHOD attribute	Type	Use
(Method Security)		Additional security attributes for methods.
MDAS_STOR_EACCK	MDAS_string	Execution-access key for logical method (all replicates).
MDAS_STOR_EACCM	MDAS_integer	Execution-access key method for logical method (all replicates).

MDAS_METHOD attribute	Type	Use
(Replicate Security)		Additional security attributes for methods.
MDAS_REPL_EACCK	MDAS_string	Execution-access key for a method replicate (single instance).
MDAS_REPL_EACCM	MDAS_integer	Execution-access key method for a method replicate (single instance).

MDAS_METHOD attribute	Type	Use
(Method Inputs)		
MDAS_METH_INC	MDAS_integer	# of inputs (data set inputs) to method. May be used as search criteria across replicate instances of methods.
MDAS_METH_FIN	MDAS_integer array	Format Catalog id#'s of inputs to method. Dimension is MDAS_METH_INC. May be used as search criteria across replicate instances of methods.
MDAS_METH_DIN	MDAS_integer array	Data set Catalog id#'s of any *static* inputs to method. This array identifies fixed data sets that must be used each time the method is executed. $MDAS\_METH\_DIN_i$ = NULL indicates no fixed data set constraint for input i. Dimension is MDAS_METH_INC. May be used as search criteria across replicate instances of methods.
MDAS_METH_RIN	MDAS_integer array	Resource Catalog id#'s of any *static* resource constraints on method inputs. This array identifies any fixed resources that must be used with particular inputs. $MDAS\_METH\_RIN_i$ = NULL indicates no fixed resource constraint for input i. Dimension is MDAS_METH_INC. May be used as search criteria across replicate instances of methods.
MDAS_METH_UIN	MDAS_integer array	User Catalog id#'s of any *static* user constraints on method inputs. This array identifies any fixed users that must have ownership of particular inputs. $MDAS\_METH\_RIN_i$ = NULL indicates no fixed user constraint for input i. Dimension is MDAS_METH_INC. May be used as search criteria across replicate instances of methods.
MDAS_METH_PIN	MDAS_integer array	The rank of input $\#i$ in the method parameter specification MDAS_METH_PARAM. Dimension is MDAS_METH_INC. May be used as search criteria across replicate instances of methods.

MDAS_METHOD attribute	Type	Use
(Method Repl. Inputs)		
MDAS_REPL_INC	MDAS_integer	# of inputs (data set inputs) to method instance.
MDAS_REPL_FIN	MDAS_integer array	Format Catalog id#'s of inputs to method instance. Dimension is MDAS_REPL_INC. Array for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_DIN	MDAS_integer array	Data set Catalog id#'s of any *static* inputs to method instance. This array identifies fixed data sets that must be used each time the method is executed. MDAS_REPL_DIN_i = NULL indicates no fixed data set constraint for input i. Dimension is MDAS_REPL_INC. Array for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_RIN	MDAS_integer array	Resource Catalog id#'s of any *static* resource constraints on method inputs. This array identifies any fixed resources that must be used with particular inputs. MDAS_REPL_RIN_i = NULL indicates no fixed resource constraint for input i. Dimension is MDAS_REPL_INC. Array for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_UIN	MDAS_integer array	User Catalog id#'s of any *static* user constraints on method inputs. This array identifies any fixed users that must have ownership of particular inputs. MDAS_REPL_RIN_i = NULL indicates no fixed user constraint for input i. Dimension is MDAS_REPL_INC. Array for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_PIN	MDAS_integer array	The rank of input $\#i$ in the method parameter specification MDAS_REPL_PARAM. Dimension is MDAS_REPL_INC. Array for replicate r is read with MDAS_INFO_SCAN_RATTR().

MDAS_METHOD attribute	Type	Use
(Method Outputs) MDAS_METH_OUTC	MDAS_integer	# of outputs (data set outputs) from method. May be used as search criteria across replicate instances of methods.
MDAS_METH_FOUT	MDAS_integer array	Format Catalog id#'s of outputs from method. Dimension is MDAS_METH_OUTC. May be used as search criteria across replicate instances of methods.
MDAS_METH_DOUT	MDAS_integer array	Data set Catalog id#'s of any *static* outputs from method. This array identifies fixed data sets that are reproduced each time the method is executed. MDAS_METH_DOUT_i = NULL indicates no fixed data set constraint for output i. Dimension is MDAS_METH_OUTC. May be used as search criteria across replicate instances of methods.
MDAS_METH_ROUT	MDAS_integer array	Resource Catalog id#'s of any *static* resource constraints on method outputs. This array identifies any fixed resources that must be used with particular outputs. MDAS_METH_ROUT_i = NULL indicates no fixed resource constraint for output i. Dimension is MDAS_METH_OUTC. May be used as search criteria across replicate instances of methods.
MDAS_METH_UOUT	MDAS_integer array	User Catalog id#'s of any *static* user constraints on method outputs. This array identifies any fixed users that must have ownership of particular outputs. MDAS_METH_ROUT_i = NULL indicates no fixed user constraint for output i. Dimension is MDAS_METH_OUTC. May be used as search criteria across replicate instances of methods.
MDAS_METH_POUT	MDAS_integer array	The rank of output $\#i$ in the method parameter specification MDAS_METH_PARAM. Dimension is MDAS_METH_OUTC. May be used as search criteria across replicate instances of methods.

MDAS_METHOD attribute	Type	Use
(Method Repl. Outputs) MDAS_REPL_OUTC	MDAS_integer	# of outputs (data set outputs) from method. Value for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_FOUT	MDAS_integer array	Format Catalog id#'s of outputs from method. Dimension is MDAS_REPL_OUTC. Array for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_DOUT	MDAS_integer array	Data set Catalog id#'s of any *static* outputs from method. This array identifies fixed data sets that are reproduced each time the method is executed. MDAS_REPL_DOUT_i = NULL indicates no fixed data set constraint for output i. Dimension is MDAS_REPL_OUTC. Array for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_ROUT	MDAS_integer array	Resource Catalog id#'s of any *static* resource constraints on method outputs. This array identifies any fixed resources that must be used with particular outputs. MDAS_REPL_ROUT_i = NULL indicates no fixed resource constraint for output i. Dimension is MDAS_REPL_OUTC. Array for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_UOUT	MDAS_integer array	User Catalog id#'s of any *static* user constraints on method outputs. This array identifies any fixed users that must have ownership of particular outputs. MDAS_REPL_ROUT_i = NULL indicates no fixed user constraint for output i. Dimension is MDAS_REPL_OUTC. Array for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_POUT	MDAS_integer array	The rank of output $\#i$ in the method parameter specification MDAS_REPL_PARAM. Dimension is MDAS_REPL_OUTC. Array for replicate r is read with MDAS_INFO_SCAN_RATTR().

MDAS_METHOD attribute	Type	Use
(Execution Parameters) MDAS_METH_PTMPL	MDAS_integer	Parameter template for method. Utilizes numeric dummy variables for input and output data sets. Dummy value 0 is reserved for the storage name of the executable in the spirit of Unix `argv[0]`. All other dummy values are unique and greater than 0. MDAS_METH_PIN contains the mapping of listed inputs to dummy values for inputs. MDAS_METH_POUT contains the mapping of listed inputs to dummy values for outputs. NULL values in either array indicate no mapping. Note that MDAS_METH_PIN and MDAS_METH_POUT are disjoint with the possible exception of NULL values. May be used as search criteria across replicate instances of methods.
MDAS_METH_PFMT	MDAS_integer	Format Catalog id# parameter template. May be used as search criteria across replicate instances of methods.
MDAS_METH_PMETH	MDAS_integer	Method Catalog id# for method (or MDAS Library routine) which takes MDAS_METH_PIN, MDAS_METH_POUT, MDAS_METH_PTMPL, MDAS_STOR_DIR, MDAS_STOR_NAME, etc. as input and produces a parameter stream suitable for MDAS_METH_EXECS. May be used as search criteria across replicate instances of methods.
MDAS_METH_EXECS	MDAS_integer	Server Catalog id# for service (or MDAS Library routine) which takes output of MDAS_METH_PMETH and executes the MDAS_METHOD. May be used as search criteria across replicate instances.

MDAS_METHOD attribute	Type	Use
(Repl. Execution Param's) MDAS_REPL_PTMPL	MDAS_integer	Parameter template for method instance. Utilizes numeric dummy variables for input and output data sets. Dummy value 0 is reserved for the storage name of the executable in the spirit of Unix `argv[0]`. All other dummy values are unique and greater than 0. MDAS_REPL_PIN contains the mapping of listed inputs to dummy values for inputs. MDAS_REPL_POUT contains the mapping of listed inputs to dummy values for outputs. NULL values in either array indicate no mapping. Note that MDAS_REPL_PIN and MDAS_REPL_POUT are disjoint with the possible exception of NULL values. Value for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_PFMT	MDAS_integer	Format Catalog id# parameter template for method instance. Value for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_PREPL	MDAS_integer	Method Catalog id# for method (or MDAS Library routine) which takes MDAS_REPL_PIN, MDAS_REPL_POUT, MDAS_REPL_PTMPL, MDAS_REPL_DIR, MDAS_REPL_NAME, etc. as input and produces a parameter stream suitable for MDAS_REPL_EXECS. Value for replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_EXECS	MDAS_integer	Server Catalog id# for service (or MDAS Library routine) which takes output of MDAS_REPL_PREPL and executes the MDAS_METHOD instance. Value for replicate r is read with MDAS_INFO_SCAN_RATTR().

MDAS_METHOD attribute	Type	Use
(Method Flow)		Methods may be coupled together to form *flows*. A flow is a compound method with specified linkage between its input/output and the inputs/outputs of the member methods, plus specific internal linkages between the inputs and outputs of the members.
MDAS_METH_FLOWC	MDAS_integer	# of methods forming flow. May be used as search criteria across replicate instances of methods.
MDAS_METH_FLOWM	MDAS_integer array	Method Catalog id#'s of flow methods. Dimension is MDAS_METH_FLOWC. May be used as search criteria across replicate instances of methods.
MDAS_METH_FLOWI	MDAS_integer array	Linkage of flow inputs to each method input. Dimension is 2 by MDAS_METH_FLOWC. If $\text{MDAS\_METH\_FLOWI}_{i1} = m$ and $\text{MDAS\_METH\_FLOWI}_{i2} = k$ then input i of the flow maps to input k of flow method m. May be used as search criteria across replicate instances of methods.
MDAS_METH_FLOWO	MDAS_integer array	Linkage of flow outputs to each method output. Dimension is 2 by MDAS_METH_FLOWC. If $\text{MDAS\_METH\_FLOWO}_{j1} = n$ and $\text{MDAS\_METH\_FLOWO}_{j2} = m$ then input j of the flow maps to input n of flow method m. May be used as search criteria across replicate instances of methods.
MDAS_METH_FLOWE	MDAS_integer matrix	Execution dependence matrix. Dimension is MDAS_METH_FLOWC by MDAS_METH_FLOWC. If $\text{MDAS\_METH\_FLOWE}_{st}$ = true then the execution of flow method s depends (in some unspecified way) on the execution of flow method t. If false, then there is no dependency of s on t. If the value is MDAS_RECUR, then s has recursive dependencies on t. Flows with recursion will be executed at least two times until a first version of the final output(s) defined in MDAS_METH_FLOWO are produced. Note that input and output linkage is given in MDAS_METH_FLOWI and MDAS_METH_FLOWO. May be used as search criteria across replicate instances of methods.

MDAS_METHOD attribute	Type	Use
(Flow Instances)		Methods may be coupled together to form *flows*. A flow is a compound method with specified linkage between its input/output and the inputs/outputs of the member methods, plus specific internal linkages between the inputs and outputs of the members.
MDAS_REPL_FLOWC	MDAS_integer	# of methods forming flow. Value for flow replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_FLOWM	MDAS_integer array	Method Catalog id#'s and replicate #'s of flow instance methods; i.e., the constituent methods of a particular flow replicate. Dimension is 2 by MDAS_REPL_FLOWC. $\text{MDAS\_REPL\_FLOWM}_{i1}$ is flow instance method, $\text{MDAS\_REPL\_FLOWM}_{i2}$ is specific replicate of flow instance method. Array for flow replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_FLOWI	MDAS_integer array	Linkage of flow inputs to each method input. Dimension is 3 by MDAS_REPL_FLOWC. If $\text{MDAS\_REPL\_FLOWI}_{i1} = m$ and $\text{MDAS\_REPL\_FLOWI}_{i2} = k$ then input i of the flow maps to input k of flow method m. May be used as search criteria across replicate instances of methods. Array for flow replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_FLOWO	MDAS_integer array	Linkage of flow outputs to each method output. Dimension is 2 by MDAS_REPL_FLOWC. If $\text{MDAS\_METH\_FLOWO}_{j1} = n$, $\text{MDAS\_METH\_FLOWO}_{j2} = m$, and $\text{MDAS\_METH\_FLOWO}_{j3} = r$ then output j of the flow maps to output n of replicate r of flow method m. Array for flow replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_FLOWE	MDAS_integer matrix	Execution dependence matrix. Dimension is MDAS_REPL_FLOWC by MDAS_REPL_FLOWC. If $\text{MDAS\_REPL\_FLOWE}_{st}$ = true then the execution of flow method s depends (in some unspecified way) on the execution of flow method t. If false, then there is no dependency of s on t. If the value is MDAS_RECUR, then s has recursive dependencies on t. Flows with recursion will be re-executed until the first version(s) of the final output(s) defined in MDAS_REPL_FLOWO are produced. Note that input and output linkage is given in MDAS_REPL_FLOWI and MDAS_REPL_FLOWO. Array for flow replicate r is read with MDAS_INFO_SCAN_RATTR().

MDAS_METHOD attribute	Type	Use
(Intra-Flow Linkage)		Methods may be coupled together to form *flows*. A flow is a compound method with specified linkage between its input/output and the inputs/outputs of the member methods, plus specific internal linkages between the inputs and outputs of the members.
MDAS_REPL_FLOWX	MDAS_integer array	For each MDAS_REPL_FLOWM, the # of inputs with intra-flow dependencies. An input has intra-flow dependency if it requires the output of another method in the flow. MDAS_REPL_FLOWX_i is the # of dependent inputs for replicate $\text{MDAS\_REPL\_FLOWM}_{i2}$ of method $\text{MDAS\_REPL\_FLOWM}_{i1}$. Array for flow replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_FLOWU	MDAS_integer array	For each MDAS_REPL_FLOWM dependent input, the dependent output and method #. $\text{MDAS\_REPL\_FLOWU}_{i1}$ is the of dependent input # in method i, $\text{MDAS\_REPL\_FLOWU}_{i2}$ is the flow method instance in MDAS_REPL_FLOWM on which method i depends, and $\text{MDAS\_REPL\_FLOWU}_{i3}$ is the output # of $\text{MDAS\_REPL\_FLOWU}_{i2}$ which input $\text{MDAS\_REPL\_FLOWU}_{i1}$ needs. Array for flow replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_FLOWY	MDAS_integer array	For each MDAS_REPL_FLOWM, the # of outputs with intra-flow dependencies. An output has intra-flow dependency if it is required for input by of another method in the flow. MDAS_REPL_FLOWY_i is the # of dependent outputs for replicate $\text{MDAS\_REPL\_FLOWM}_{i2}$ of method $\text{MDAS\_REPL\_FLOWM}_{i1}$. Array for flow replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_FLOWV	MDAS_integer array	For each MDAS_REPL_FLOWM dependent output, the dependent input and method #. $\text{MDAS\_REPL\_FLOWU}_{i1}$ is the # of the dependent output in method i, $\text{MDAS\_REPL\_FLOWU}_{i2}$ is the flow method instance in MDAS_REPL_FLOWM which is dependent upon method i, and $\text{MDAS\_REPL\_FLOWU}_{i3}$ is the input # in $\text{MDAS\_REPL\_FLOWU}_{i2}$ which needs $\text{MDAS\_REPL\_FLOWU}_{i1}$. Array for flow replicate r is read with MDAS_INFO_SCAN_RATTR().
MDAS_REPL_FLOWZ	MDAS_integer matrix	Data flow dependence matrix. The inputs to each method in MDAS_REPL_FLOWM form a block of rows in MDAS_REPL_FLOWZ, there are a total of Σ MDAS_REPL_FLOWX_i rows. Similarly, the outputs of each flow method form a block of columns and there are Σ MDAS_REPL_FLOWY_j columns. If $\text{MDAS\_REPL\_FLOWX}_{ij}$ = MDAS_TRUE then output $\#j$ is linked to input $\#i$. Matrix values are otherwise false. If an entire column is false and the corresponding output data set is new, it may be discarded. Matrix for flow replicate r is read with MDAS_INFO_SCAN_RATTR().

Shared Attributes

MDAS_METHOD attribute	Type	Use
(Base)		Methods share these attributes with data sets.
MDAS_NAME	MDAS_string	
MDAS_ID	MDAS_integer	
MDAS_CD_ID	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Version)		Methods share these attributes with data sets.
MDAS_VERSION	MDAS_string	
MDAS_VERSIONX	MDAS_logical	
MDAS_VERSIONM	MDAS_integer	
MDAS_VERSIONP	MDAS_integer	
MDAS_VERSIONN	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Alias)		Methods share these attributes with data sets.
MDAS_ALIAS	MDAS_string	
MDAS_ALIASC	MDAS_integer	
MDAS_ALIASV	MDAS_string array	

MDAS_METHOD attribute	Type	Use
(Documentation)		Methods share these attributes with data sets.
MDAS_ABSTRACT	MDAS_integer	
MDAS_DOC	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(SD)		Methods share these attributes with data sets.
MDAS_SD_TABLE	MDAS_integer	
MDAS_SD_KEYC	MDAS_integer	
MDAS_SD_COLS	MDAS_string array	
MDAS_SD_KEY_TYP	MDAS_integer	
MDAS_SD_KEYS	MDAS_handle	
MDAS_SD_LOB_COL	MDAS_string	

MDAS_METHOD attribute	Type	Use
(Logical Group)		Methods share these attributes with data sets.
MDAS_STOR_GRPN	MDAS_string	
MDAS_STOR_GRPI	MDAS_integer	
MDAS_STOR_GRPNC	MDAS_integer	
MDAS_STOR_GRPNV	MDAS_string array	
MDAS_STOR_GRPIC	MDAS_integer	
MDAS_STOR_GRPIV	MDAS_integer array	

MDAS_METHOD attribute	Type	Use
(Logical Domain)		Methods share these attributes with data sets.
MDAS_STOR_DMNN	MDAS_string	
MDAS_STOR_DMNI	MDAS_integer	
MDAS_STOR_DMNNC	MDAS_integer	
MDAS_STOR_DMNNV	MDAS_string array	
MDAS_STOR_DMNIC	MDAS_integer	
MDAS_STOR_DMNIV	MDAS_integer array	

MDAS_METHOD attribute	Type	Use
(Input Lineage)		Methods share these attributes with data sets.
MDAS_STOR_IN	MDAS_integer	
MDAS_STOR_INC	MDAS_integer	
MDAS_STOR_INV	MDAS_integer array	

MDAS_METHOD attribute	Type	Use
(Consequential Lineage)		Methods share these attributes with data sets.
MDAS_STOR_OUT	MDAS_integer	
MDAS_STOR_OUTC	MDAS_integer	
MDAS_STOR_OUTV	MDAS_integer array	

MDAS_METHOD attribute	Type	Use
(Generation Lineage)		Methods share these attributes with data sets.
MDAS_STOR_GEN	MDAS_integer	
MDAS_STOR_GNP	MDAS_string	
MDAS_STOR_GNR	MDAS_integer	
MDAS_STOR_GNU	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Re-Generation Policy)		Methods share these attributes with data sets.
MDAS_STOR_PLCY	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Trigger)		Methods share these attributes with data sets.
MDAS_STOR_TRGM	MDAS_integer	
MDAS_STOR_TRGP	MDAS_integer	
MDAS_STOR_TRGC	MDAS_integer	
MDAS_STOR_TRGMV	MDAS_integer array	
MDAS_STOR_TRGPV	MDAS_integer array	

MDAS_METHOD attribute	Type	Use
(Storage Date)		Methods share these attributes with data sets.
MDAS_STOR_DATE	MDAS_time	

MDAS_METHOD attribute	Type	Use
(Storage Permanence)		Methods share these attributes with data sets.
MDAS_STOR_PERM	MDAS_double	
MDAS_STOR_PURG	MDAS_logical	

MDAS_METHOD attribute	Type	Use
(Storage Size)		Methods share these attributes with data sets.
MDAS_STOR_SIZE	MDAS_size	

MDAS_METHOD attribute	Type	Use
(Storage Format)		Methods share these attributes with data sets.
MDAS_STOR_FMTN	MDAS_string	
MDAS_STOR_FMTI	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Storage Resource)		Methods share these attributes with data sets.
MDAS_STOR_RSCN	MDAS_string	
MDAS_STOR_RSCI	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Storage Server)		Methods share these attributes with data sets.
MDAS_STOR_SRVN	MDAS_string	
MDAS_STOR_SRVI	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Storage Path and Name)		Methods share these attributes with data sets.
MDAS_STOR_DIR	MDAS_string	
MDAS_STOR_NAM	MDAS_string	

MDAS_METHOD attribute	Type	Use
(Method Owner)		Methods share these attributes with data sets.
MDAS_STOR_OWN	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Method SpecHist)		Methods share these attributes with data sets.
MDAS_STOR_HSA	MDAS_integer	
MDAS_STOR_HST	MDAS_time	
MDAS_STOR_HSS	MDAS_spectrum	
MDAS_STOR_HSC	MDAS_integer	
MDAS_STOR_HSAV	MDAS_integer array	
MDAS_STOR_HSTV	MDAS_time array	
MDAS_STOR_HSSV	MDAS_spectrum array	

MDAS_METHOD attribute	Type	Use
(Method Perf)		Methods share these attributes with data sets.
MDAS_STOR_PERF	TBD	TBD.

MDAS_METHOD attribute	Type	Use
(Method Lock)		Methods share these attributes with data sets.
MDAS_STOR_RLCK	MDAS_logical	
MDAS_STOR_RLCKS	MDAS_time	
MDAS_STOR_RLCKE	MDAS_time	
MDAS_STOR_RLCKD	MDAS_integer	
MDAS_STOR_WLCK	MDAS_logical	
MDAS_STOR_WLCKS	MDAS_time	
MDAS_STOR_WLCKE	MDAS_time	
MDAS_STOR_WLCKD	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Method Security)		Methods share these attributes with data sets.
MDAS_STOR_AUTK	MDAS_string	
MDAS_STOR_AUTM	MDAS_integer	
MDAS_STOR_RACCK	MDAS_string	
MDAS_STOR_RACCM	MDAS_integer	
MDAS_STOR_WACCK	MDAS_string	
MDAS_STOR_WACCM	MDAS_integer	
MDAS_STOR_CRYK	MDAS_string	
MDAS_STOR_CRYM	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Replicates)		Methods share these attributes with data sets.
MDAS_REPLICATES	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Replicate Group)		Methods share these attributes with data sets.
MDAS_REPL_GRPN	MDAS_string	
MDAS_REPL_GRPNC	MDAS_integer array	
MDAS_REPL_GRPNV	MDAS_string array	
MDAS_REPL_GRPI	MDAS_integer	
MDAS_REPL_GRPIC	MDAS_integer array	
MDAS_REPL_GRPIV	MDAS_string array	

MDAS_METHOD attribute	Type	Use
(Replicate Domain)		Methods share these attributes with data sets.
MDAS_REPL_DMNN	MDAS_string	
MDAS_REPL_DMNNC	MDAS_integer array	
MDAS_REPL_DMNNV	MDAS_string array	
MDAS_REPL_DMNI	MDAS_integer	
MDAS_REPL_DMNIC	MDAS_integer array	
MDAS_REPL_DMNIV	MDAS_string array	

MDAS_METHOD attribute	Type	Use
(Repl. Input Lineage)		Methods share these attributes with data sets.
MDAS_REPL_IN	MDAS_integer	
MDAS_REPL_INC	MDAS_integer	
MDAS_REPL_INV	MDAS_integer array	

MDAS_METHOD attribute	Type	Use
(Repl. Conseq. Lineage)		Methods share these attributes with data sets.
MDAS_REPL_OUT	MDAS_integer	
MDAS_REPL_OUTVC	MDAS_integer	
MDAS_REPL_OUTV	MDAS_integer array	

MDAS_METHOD attribute	Type	Use
(Repl. Gen. Lineage)		Methods share these attributes with data sets.
MDAS_REPL_GEN	MDAS_integer	
MDAS_REPL_GNP	MDAS_string	
MDAS_REPL_GNR	MDAS_integer	
MDAS_REPL_GNU	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Replication Trigger)		Methods share these attributes with data sets.
MDAS_REPL_TRGM	MDAS_integer	
MDAS_REPL_TRGP	MDAS_integer	
MDAS_REPL_TRGC	MDAS_integer	
MDAS_REPL_TRGMV	MDAS_integer array	
MDAS_REPL_TRGPV	MDAS_integer array	

MDAS_METHOD attribute	Type	Use
(Repl. Re-Gen. Policy)		Methods share these attributes with data sets.
MDAS_REPL_PLCY	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Replicate Dates)		Methods share these attributes with data sets.
MDAS_REPL_DATE	MDAS_time	
MDAS_REPL_DATEV	MDAS_time array	

MDAS_METHOD attribute	Type	Use
(Replicate Permanence)		Methods share these attributes with data sets.
MDAS_REPL_PERM	MDAS_double	
MDAS_REPL_PERMV	MDAS_double array	
MDAS_REPL_PURG	MDAS_logical	
MDAS_REPL_PURGV	MDAS_logical array	

MDAS_METHOD attribute	Type	Use
(Replicate Sizes)		Methods share these attributes with data sets.
MDAS_REPL_SIZE	MDAS_size	
MDAS_REPL_SIZEV	MDAS_size array	

MDAS_METHOD attribute	Type	Use
(Replicate Format)		Methods share these attributes with data sets.
MDAS_REPL_FMTH	MDAS_logical	
Note: a method replicate with heterogeneous format segments will not have any data in the format attributes below. See MDAS_REPL_SEGC.		
MDAS_REPL_FMTN	MDAS_string	
MDAS_REPL_FMTNV	MDAS_string array	
MDAS_REPL_FMTI	MDAS_integer	
MDAS_REPL_FMTIV	MDAS_integer array	

MDAS_METHOD attribute	Type	Use
(Repl. Storage Resource)		Methods share these attributes with data sets.
MDAS_REPL_RSCH	MDAS_logical	
Note: a method replicate with distributed storage segments will not have any data in the resource attributes below. See MDAS_REPL_SEGC.		
MDAS_REPL_RSCN	MDAS_string	
MDAS_REPL_RSCNV	MDAS_string array	
MDAS_REPL_RSCI	MDAS_integer	
MDAS_REPL_RSCIV	MDAS_integer array	

MDAS_METHOD attribute	Type	Use
(Repl. Storage Server)		Methods share these attributes with data sets.
MDAS_REPL_SRVH	MDAS_logical	
Note: a method replicate with heterogeneous storage service segments will not have any data in the server attributes below. See MDAS_REPL_SEGC.		
MDAS_REPL_SRVN	MDAS_string	
MDAS_REPL_SRVNV	MDAS_string array	
MDAS_REPL_SRVI	MDAS_integer	
MDAS_REPL_SRVIV	MDAS_integer array	

MDAS_METHOD attribute	Type	Use
(Repl. Dir. and Name) Note: a method replicate with heterogeneous storage segments will not have any data in the directory and name attributes below. See MDAS_REPL_SEGC.		Methods share these attributes with data sets.
MDAS_REPL_DIRN	MDAS_string	
MDAS_REPL_DIRNV	MDAS_string array	
MDAS_REPL_NAMN	MDAS_string	
MDAS_REPL_NAMNV	MDAS_string array	

MDAS_METHOD attribute	Type	Use
(Replicate Owner)		Methods share these attributes with data sets.
MDAS_REPL_OWN	MDAS_string	
MDAS_REPL_OWNV	MDAS_string array	

MDAS_METHOD attribute	Type	Use
(Replicate SpecHist)		Methods share these attributes with data sets.
MDAS_REPL_HSA	MDAS_integer	
MDAS_REPL_HST	MDAS_time	
MDAS_REPL_HSS	MDAS_spectrum	
MDAS_REPL_HSC	MDAS_integer	
MDAS_REPL_HSAV	MDAS_integer array	
MDAS_REPL_HSTV	MDAS_time array	
MDAS_REPL_HSSV	MDAS_spectrum array	

MDAS_METHOD attribute	Type	Use
(Replicate Lock)		Methods share these attributes with data sets.
MDAS_REPL_RLCK	MDAS_logical	
MDAS_REPL_RLCKS	MDAS_time	
MDAS_REPL_RLCKE	MDAS_time	
MDAS_REPL_RLCKD	MDAS_integer	
MDAS_REPL_WLCK	MDAS_logical	
MDAS_REPL_WLCKS	MDAS_time	
MDAS_REPL_WLCKE	MDAS_time	
MDAS_REPL_WLCKD	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Replicate Security)		Methods share these attributes with data sets.
MDAS_REPL_AUTK	MDAS_string	
MDAS_REPL_AUTM	MDAS_integer	
MDAS_REPL_RACCK	MDAS_string	
MDAS_REPL_RACCM	MDAS_integer	
MDAS_REPL_WACCK	MDAS_string	
MDAS_REPL_WACCM	MDAS_integer	
MDAS_REPL_CRYK	MDAS_string	
MDAS_REPL_CRYM	MDAS_integer	

MDAS_METHOD attribute	Type	Use
(Replicate Perf)		Methods share these attributes with data sets.
MDAS_REPL_PERF	TBD	TBD.

MDAS_METHOD attribute	Type	Use
(Segment Storage)		Methods share these attributes with data sets.
MDAS_REPL_SEGC	MDAS_integer	
MDAS_REPL_SEGM	MDAS_integer	
MDAS_REPL_SEGRV	MDAS_integer array	
MDAS_REPL_SEGIV	MDAS_integer array	

4.7.2.2.3 MDAS_RESOURCE

Resources share many attributes with data sets and methods. Since resources are not truly "stored", the prefix `MDAS_STOR_` is redefined to `MDAS_RSRC` for resources. Further, resources are not considered to be replicable. To identify resources with similar (or identical) characteristics, define a group and give the resources membership in that group.

Here, all attributes of resources are listed for completeness. Descriptions are only given for those attributes unique to resources.

A resource *location* is a description of its network address and physical local.

A resource *segment* is a physical component (e.g., peripheral) of a compound computer system.

Unique Resource Attributes

MDAS_RESOURCE attribute	Type	Use
(Resource Location) MDAS_RSRC_LOC	TBD	TBD.

MDAS_RESOURCE attribute	Type	Use
(Resource Services)		
MDAS_RSRC_SRVI	MDAS_integer	Catalog id# of some available server.
MDAS_RSRC_SRVN	MDAS_integer	Name or alias of some available server.
MDAS_RSRC_SRVC	MDAS_integer	# of services available.
MDAS_RSRC_SRVIV	MDAS_integer array	Catalog id#'s of available servers.
MDAS_RSRC_SRVNV	MDAS_string array	Names of available servers.

Shared Attributes

MDAS_RESOURCE attribute	Type	Use
(Base)		Resources share these attributes with data sets.
MDAS_NAME	MDAS_string	
MDAS_ID	MDAS_integer	
MDAS_CD_ID	MDAS_integer	

MDAS_RESOURCE attribute	Type	Use
(Version)		Resources share these attributes with data sets.
MDAS_VERSION	MDAS_string	
MDAS_VERSIONX	MDAS_logical	
MDAS_VERSIONM	MDAS_integer	
MDAS_VERSIONP	MDAS_integer	
MDAS_VERSIONN	MDAS_integer	

MDAS_RESOURCE attribute	Type	Use
(Alias)		Resources share these attributes with data sets.
MDAS_ALIAS	MDAS_string	
MDAS_ALIASC	MDAS_integer	
MDAS_ALIASV	MDAS_string array	

MDAS_RESOURCE attribute	Type	Use
(Documentation)		Resources share these attributes with data sets.
MDAS_ABSTRACT	MDAS_integer	
MDAS_DOC	MDAS_integer	

MDAS_RESOURCE attribute	Type	Use
(Logical Group)		Resources share these attributes with data sets.
MDAS_RSRC_GRPN	MDAS_string	
MDAS_RSRC_GRPI	MDAS_integer	
MDAS_RSRC_GRPNC	MDAS_integer	
MDAS_RSRC_GRPNV	MDAS_string array	
MDAS_RSRC_GRPIC	MDAS_integer	
MDAS_RSRC_GRPIV	MDAS_integer array	

MDAS_RESOURCE attribute	Type	Use
(Logical Domain)		Resources share these attributes with data sets.
MDAS_RSRC_DMNN	MDAS_string	
MDAS_RSRC_DMNI	MDAS_integer	
MDAS_RSRC_DMNNC	MDAS_integer	
MDAS_RSRC_DMNNV	MDAS_string array	
MDAS_RSRC_DMNIC	MDAS_integer	
MDAS_RSRC_DMNIV	MDAS_integer array	

MDAS_RESOURCE attribute	Type	Use
(Trigger)		Resources share these attributes with data sets.
MDAS_RSRC_TRGM	MDAS_integer	
MDAS_RSRC_TRGP	MDAS_integer	
MDAS_RSRC_TRGC	MDAS_integer	
MDAS_RSRC_TRGMV	MDAS_integer array	
MDAS_RSRC_TRGPV	MDAS_integer array	

MDAS_RESOURCE attribute	Type	Use
(Instantiation Date)		Resources share these attributes with data sets.
MDAS_RSRC_DATE	MDAS_time	

MDAS_RESOURCE attribute	Type	Use
(Resource Format)		Resources share these attributes with data sets.
MDAS_RSRC_FMTN	MDAS_string	
MDAS_RSRC_FMTI	MDAS_integer	

MDAS_RESOURCE attribute	Type	Use
(Resource Owner)		Resources share these attributes with data sets.
MDAS_RSRC_OWN	MDAS_integer	

MDAS_RESOURCE attribute	Type	Use
(Resource SpecHist)		Resources share these attributes with data sets.
MDAS_RSRC_HSA	MDAS_integer	
MDAS_RSRC_HST	MDAS_time	
MDAS_RSRC_HSS	MDAS_spectrum	
MDAS_RSRC_HSC	MDAS_integer	
MDAS_RSRC_HSAV	MDAS_integer array	
MDAS_RSRC_HSTV	MDAS_time array	
MDAS_RSRC_HSSV	MDAS_spectrum array	

MDAS_RESOURCE attribute	Type	Use
(Resource Perf)		Resources share these attributes with data sets.
MDAS_RSRC_PERF	TBD	TBD.

MDAS_RESOURCE attribute	Type	Use
(Resource Lock)		Resources share these attributes with data sets and methods.
MDAS_RSRC_RLCK	MDAS_logical	
MDAS_RSRC_RLCKS	MDAS_time	
MDAS_RSRC_RLCKE	MDAS_time	
MDAS_RSRC_RLCKD	MDAS_integer	
MDAS_RSRC_WLCK	MDAS_logical	
MDAS_RSRC_WLCKS	MDAS_time	
MDAS_RSRC_WLCKE	MDAS_time	
MDAS_RSRC_WLCKD	MDAS_integer	
MDAS_RSRC_ELCK	MDAS_logical	
MDAS_RSRC_ELCKS	MDAS_time	
MDAS_RSRC_ELCKE	MDAS_time	
MDAS_RSRC_ELCKD	MDAS_integer	

MDAS_RESOURCE attribute	Type	Use
(Resource Security)		Resources share these attributes with data sets and methods.
MDAS_RSRC_AUTK	MDAS_string	
MDAS_RSRC_AUTM	MDAS_integer	
MDAS_RSRC_RACCK	MDAS_string	
MDAS_RSRC_RACCM	MDAS_integer	
MDAS_RSRC_WACCK	MDAS_string	
MDAS_RSRC_WACCM	MDAS_integer	
MDAS_RSRC_EACCK	MDAS_string	
MDAS_RSRC_EACCM	MDAS_integer	
MDAS_RSRC_CRYK	MDAS_string	
MDAS_RSRC_CRYM	MDAS_integer	

MDAS_RESOURCE attribute	Type	Use
(Resource Segments)		Resources share these attributes with data sets.
MDAS_RSRC_SEGC	MDAS_integer	
MDAS_RSRC_SEGM	MDAS_integer	
MDAS_RSRC_SEGRV	MDAS_integer array	
MDAS_RSRC_SEGIV	MDAS_integer array	

4.7.2.2.4 MDAS_USER

A *user* is the owner of a data set, method, resource, or process. User *accounts* are instantiated by a system administrator and gain access to resources through membership in security domains. An individual (human) with multiple accounts is considered a single user with replicate accounts. Each replicate can have separable access characteristics through individual domain attributes. To identify users with similar (or identical) characteristics, users membership in a group or domain.

User metadata have some attributes common to data sets and methods, but are not partitioned into segments. Here, all attributes of users are listed for completeness. Descriptions are only given for those attributes unique to users.

Unique User Attributes

TBD.

Shared Attributes

MDAS_USER attribute	Type	Use
(Account Lock)		Users share these attributes with other entities.
MDAS_USER_ELCK	MDAS_logical	Users share these attributes with other entities.
MDAS_USER_ELCKS	MDAS_time	Users share these attributes with other entities.
MDAS_USER_ELCKE	MDAS_time	Users share these attributes with other entities.
MDAS_USER_ELCKD	MDAS_integer	Users share these attributes with other entities.

MDAS_USER attribute	Type	Use
(Account Lock)		Users share these attributes with other entities.
MDAS_REPL_ELCK	MDAS_logical	Users share these attributes with other entities.
MDAS_REPL_ELCKS	MDAS_time	Users share these attributes with other entities.
MDAS_REPL_ELCKE	MDAS_time	Users share these attributes with other entities.
MDAS_REPL_ELCKD	MDAS_integer	Users share these attributes with other entities.

MDAS_USER attribute	Type	Use
(Account Security)		Users share these attributes with other entities.
MDAS_STOR_EACCK	MDAS_string	Users share these attributes with other entities.
MDAS_STOR_EACCM	MDAS_integer	Users share these attributes with other entities.

MDAS_USER attribute	Type	Use
(Replicate Security)		Users share these attributes with other entities.
MDAS_REPL_EACCK	MDAS_string	Users share these attributes with other entities.
MDAS_REPL_EACCM	MDAS_integer	Users share these attributes with other entities.

MDAS_USER attribute	Type	Use
(Base)		Users share these attributes with other entities.
MDAS_NAME	MDAS_string	
MDAS_ID	MDAS_integer	
MDAS_CD_ID	MDAS_integer	

MDAS_USER attribute	Type	Use
(Version)		Users share these attributes with other entities.
MDAS_VERSION	MDAS_string	
MDAS_VERSIONX	MDAS_logical	
MDAS_VERSIONM	MDAS_integer	
MDAS_VERSIONP	MDAS_integer	
MDAS_VERSIONN	MDAS_integer	

MDAS_USER attribute	Type	Use
(Alias)		Users share these attributes with other entities.
MDAS_ALIAS	MDAS_string	
MDAS_ALIASC	MDAS_integer	
MDAS_ALIASV	MDAS_string array	

MDAS_USER attribute	Type	Use
(Documentation)		Users share these attributes with other entities.
MDAS_ABSTRACT	MDAS_integer	
MDAS_DOC	MDAS_integer	

MDAS_USER attribute	Type	Use
(SD)		Users share these attributes with other entities.
MDAS_SD_TABLE	MDAS_integer	
MDAS_SD_KEYC	MDAS_integer	
MDAS_SD_COLS	MDAS_string array	
MDAS_SD_KEY_TYP	MDAS_integer	
MDAS_SD_KEYS	MDAS_handle	
MDAS_SD_LOB_COL	MDAS_string	

MDAS_USER attribute	Type	Use
(Logical Group)		Users share these attributes with other entities.
MDAS_STOR_GRPN	MDAS_string	
MDAS_STOR_GRPI	MDAS_integer	
MDAS_STOR_GRPNC	MDAS_integer	
MDAS_STOR_GRPNV	MDAS_string array	
MDAS_STOR_GRPIC	MDAS_integer	
MDAS_STOR_GRPIV	MDAS_integer array	

MDAS_USER attribute	Type	Use
(Logical Domain)		Users share these attributes with other entities.
MDAS_STOR_DMNN	MDAS_string	
MDAS_STOR_DMNI	MDAS_integer	
MDAS_STOR_DMNNC	MDAS_integer	
MDAS_STOR_DMNNV	MDAS_string array	
MDAS_STOR_DMNIC	MDAS_integer	
MDAS_STOR_DMNIV	MDAS_integer array	

MDAS_USER attribute	Type	Use
(Input Lineage)		Users share these attributes with other entities.
MDAS_STOR_IN	MDAS_integer	
MDAS_STOR_INC	MDAS_integer	
MDAS_STOR_INV	MDAS_integer array	

MDAS_USER attribute	Type	Use
(Consequential Lineage)		Users share these attributes with other entities.
MDAS_STOR_OUT	MDAS_integer	
MDAS_STOR_OUTC	MDAS_integer	
MDAS_STOR_OUTV	MDAS_integer array	

MDAS_USER attribute	Type	Use
(Generation Lineage)		Users share these attributes with other entities.
MDAS_STOR_GEN	MDAS_integer	
MDAS_STOR_GNP	MDAS_string	
MDAS_STOR_GNR	MDAS_integer	
MDAS_STOR_GNU	MDAS_integer	

MDAS_USER attribute	Type	Use
(Re-Generation Policy)		Users share these attributes with other entities.
MDAS_STOR_PLCY	MDAS_integer	

MDAS_USER attribute	Type	Use
(Trigger)		Users share these attributes with other entities.
MDAS_STOR_TRGM	MDAS_integer	
MDAS_STOR_TRGP	MDAS_integer	
MDAS_STOR_TRGC	MDAS_integer	
MDAS_STOR_TRGMV	MDAS_integer array	
MDAS_STOR_TRGPV	MDAS_integer array	

MDAS_USER attribute	Type	Use
(Storage Date)		Users share these attributes with other entities.
MDAS_STOR_DATE	MDAS_time	

MDAS_USER attribute	Type	Use
(Storage Permanence)		Users share these attributes with other entities.
MDAS_STOR_PERM	MDAS_double	
MDAS_STOR_PURG	MDAS_logical	

MDAS_USER attribute	Type	Use
(Storage Format)		Users share these attributes with other entities.
MDAS_STOR_FMTN	MDAS_string	
MDAS_STOR_FMTI	MDAS_integer	

MDAS_USER attribute	Type	Use
(Storage Resource)		Users share these attributes with other entities.
MDAS_STOR_RSCN	MDAS_string	
MDAS_STOR_RSCI	MDAS_integer	

MDAS_USER attribute	Type	Use
(Storage Server)		Users share these attributes with other entities.
MDAS_STOR_SRVN	MDAS_string	
MDAS_STOR_SRVI	MDAS_integer	

MDAS_USER attribute	Type	Use
(Storage Path and Name)		Users share these attributes with other entities.
MDAS_STOR_DIR	MDAS_string	
MDAS_STOR_NAM	MDAS_string	

MDAS_USER attribute	Type	Use
(Account Admin.)		Users share these attributes with other entities.
MDAS_STOR_OWN	MDAS_integer	

MDAS_USER attribute	Type	Use
(User SpecHist)		Users share these attributes with other entities.
MDAS_STOR_HSA	MDAS_integer	
MDAS_STOR_HST	MDAS_time	
MDAS_STOR_HSS	MDAS_spectrum	
MDAS_STOR_HSC	MDAS_integer	
MDAS_STOR_HSAV	MDAS_integer array	
MDAS_STOR_HSTV	MDAS_time array	
MDAS_STOR_HSSV	MDAS_spectrum array	

MDAS_USER attribute	Type	Use
(User Perf)		Users share these attributes with other entities.
MDAS_STOR_PERF	TBD	TBD.

MDAS_USER attribute	Type	Use
(User Lock)		Users share these attributes with other entities.
MDAS_STOR_RLCK	MDAS_logical	
MDAS_STOR_RLCKS	MDAS_time	
MDAS_STOR_RLCKE	MDAS_time	
MDAS_STOR_RLCKD	MDAS_integer	
MDAS_STOR_WLCK	MDAS_logical	
MDAS_STOR_WLCKS	MDAS_time	
MDAS_STOR_WLCKE	MDAS_time	
MDAS_STOR_WLCKD	MDAS_integer	

MDAS_USER attribute	Type	Use
(User Security)		Users share these attributes with other entities.
MDAS_STOR_AUTK	MDAS_string	
MDAS_STOR_AUTM	MDAS_integer	
MDAS_STOR_RACCK	MDAS_string	
MDAS_STOR_RACCM	MDAS_integer	
MDAS_STOR_WACCK	MDAS_string	
MDAS_STOR_WACCM	MDAS_integer	
MDAS_STOR_CRYK	MDAS_string	
MDAS_STOR_CRYM	MDAS_integer	

MDAS_USER attribute	Type	Use
(Replicates)		Users share these attributes with other entities.
MDAS_REPLICATES	MDAS_integer	

MDAS_USER attribute	Type	Use
(Replicate Group)		Users share these attributes with other entities.
MDAS_REPL_GRPN	MDAS_string	
MDAS_REPL_GRPNC	MDAS_integer array	
MDAS_REPL_GRPNV	MDAS_string array	
MDAS_REPL_GRPI	MDAS_integer	
MDAS_REPL_GRPIC	MDAS_integer array	
MDAS_REPL_GRPIV	MDAS_string array	

MDAS_USER attribute	Type	Use
(Replicate Domain)		Users share these attributes with other entities.
MDAS_REPL_DMNN	MDAS_string	
MDAS_REPL_DMNNC	MDAS_integer array	
MDAS_REPL_DMNNV	MDAS_string array	
MDAS_REPL_DMNI	MDAS_integer	
MDAS_REPL_DMNIC	MDAS_integer array	
MDAS_REPL_DMNIV	MDAS_string array	

MDAS_USER attribute	Type	Use
(Repl. Input Lineage)		Users share these attributes with other entities.
MDAS_REPL_IN	MDAS_integer	
MDAS_REPL_INC	MDAS_integer	
MDAS_REPL_INV	MDAS_integer array	

MDAS_USER attribute	Type	Use
(Repl. Conseq. Lineage)		Users share these attributes with other entities.
MDAS_REPL_OUT	MDAS_integer	
MDAS_REPL_OUTVC	MDAS_integer	
MDAS_REPL_OUTV	MDAS_integer array	

MDAS_USER attribute	Type	Use
(Repl. Gen. Lineage)		Users share these attributes with other entities.
MDAS_REPL_GEN	MDAS_integer	
MDAS_REPL_GNP	MDAS_string	
MDAS_REPL_GNR	MDAS_integer	
MDAS_REPL_GNU	MDAS_integer	

MDAS_USER attribute	Type	Use
(Replication Trigger)		Users share these attributes with other entities.
MDAS_REPL_TRGM	MDAS_integer	
MDAS_REPL_TRGP	MDAS_integer	
MDAS_REPL_TRGC	MDAS_integer	
MDAS_REPL_TRGMV	MDAS_integer array	
MDAS_REPL_TRGPV	MDAS_integer array	

MDAS_USER attribute	Type	Use
(Repl. Re-Gen. Policy)		Users share these attributes with other entities.
MDAS_REPL_PLCY	MDAS_integer	

MDAS_USER attribute	Type	Use
(Replicate Dates)		Users share these attributes with other entities.
MDAS_REPL_DATE	MDAS_time	
MDAS_REPL_DATEV	MDAS_time array	

MDAS_USER attribute	Type	Use
(Replicate Permanence)		Users share these attributes with other entities.
MDAS_REPL_PERM	MDAS_double	
MDAS_REPL_PERMV	MDAS_double array	
MDAS_REPL_PURG	MDAS_logical	
MDAS_REPL_PURGV	MDAS_logical array	

MDAS_USER attribute	Type	Use
(Replicate Format)		Users share these attributes with other entities.
MDAS_REPL_FMTH	MDAS_logical	
MDAS_REPL_FMTN	MDAS_string	
MDAS_REPL_FMTNV	MDAS_string array	
MDAS_REPL_FMTI	MDAS_integer	
MDAS_REPL_FMTIV	MDAS_integer array	

MDAS_USER attribute	Type	Use
(Repl. Storage Resource)		Users share these attributes with other entities.
MDAS_REPL_RSCH	MDAS_logical	
MDAS_REPL_RSCN	MDAS_string	
MDAS_REPL_RSCNV	MDAS_string array	
MDAS_REPL_RSCI	MDAS_integer	
MDAS_REPL_RSCIV	MDAS_integer array	

MDAS_USER attribute	Type	Use
(Repl. Storage Server)		Users share these attributes with other entities.
MDAS_REPL_SRVH	MDAS_logical	
MDAS_REPL_SRVN	MDAS_string	
MDAS_REPL_SRVNV	MDAS_string array	
MDAS_REPL_SRVI	MDAS_integer	
MDAS_REPL_SRVIV	MDAS_integer array	

MDAS_USER attribute	Type	Use
(Repl. Dir. and Name)		Users share these attributes with other entities.
MDAS_REPL_DIRN	MDAS_string	
MDAS_REPL_DIRNV	MDAS_string array	
MDAS_REPL_NAMN	MDAS_string	
MDAS_REPL_NAMNV	MDAS_string array	

MDAS_USER attribute	Type	Use
(Replicate Owner)		Users share these attributes with other entities.
MDAS_REPL_OWN	MDAS_string	
MDAS_REPL_OWNV	MDAS_string array	

MDAS_USER attribute	Type	Use
(Replicate SpecHist)		Users share these attributes with other entities.
MDAS_REPL_HSA	MDAS_integer	
MDAS_REPL_HST	MDAS_time	
MDAS_REPL_HSS	MDAS_spectrum	
MDAS_REPL_HSC	MDAS_integer	
MDAS_REPL_HSAV	MDAS_integer array	
MDAS_REPL_HSTV	MDAS_time array	
MDAS_REPL_HSSV	MDAS_spectrum array	

MDAS_USER attribute	Type	Use
(Replicate Lock)		Users share these attributes with other entities.
MDAS_REPL_RLCK	MDAS_logical	
MDAS_REPL_RLCKS	MDAS_time	
MDAS_REPL_RLCKE	MDAS_time	
MDAS_REPL_RLCKD	MDAS_integer	
MDAS_REPL_WLCK	MDAS_logical	
MDAS_REPL_WLCKS	MDAS_time	
MDAS_REPL_WLCKE	MDAS_time	
MDAS_REPL_WLCKD	MDAS_integer	

MDAS_USER attribute	Type	Use
(Replicate Security)		Users share these attributes with other entities.
MDAS_REPL_AUTK	MDAS_string	
MDAS_REPL_AUTM	MDAS_integer	
MDAS_REPL_RACCK	MDAS_string	
MDAS_REPL_RACCM	MDAS_integer	
MDAS_REPL_WACCK	MDAS_string	
MDAS_REPL_WACCM	MDAS_integer	
MDAS_REPL_CRYK	MDAS_string	
MDAS_REPL_CRYM	MDAS_integer	

MDAS_USER attribute	Type	Use
(Replicate Perf)		Users share these attributes with other entities.
MDAS_REPL_PERF	TBD	TBD.

4.7.2.3 Auxilary Entities

4.7.2.3.1 MDAS_FORMAT

A *format* is an MDAS token that identifies the digital or logical structure of an MDAS Entity.

A method which changes the format of an MDAS entity without changing the entity content is said to be invariant.

Format metadata have a few attributes common to data sets and methods but no storage.

Here, all attributes of formats are listed for completeness. Descriptions are only given for those attributes unique to formats.

Unique Format Attributes

TBD.

Shared Attributes

MDAS_FORMAT attribute	Type	Use
(Base)		Formats share these attributes with other entities.
MDAS_NAME	MDAS_string	
MDAS_ID	MDAS_integer	
MDAS_CD_ID	MDAS_integer	

MDAS_FORMAT attribute	Type	Use
(Version)		Formats share these attributes with other entities.
MDAS_VERSION	MDAS_string	
MDAS_VERSIONX	MDAS_logical	
MDAS_VERSIONM	MDAS_integer	
MDAS_VERSIONP	MDAS_integer	
MDAS_VERSIONN	MDAS_integer	

MDAS_FORMAT attribute	Type	Use
(Alias)		Formats share these attributes with other entities.
MDAS_ALIAS	MDAS_string	
MDAS_ALIASC	MDAS_integer	
MDAS_ALIASV	MDAS_string array	

MDAS_FORMAT attribute	Type	Use
(Documentation)		Formats share these attributes with other entities.
MDAS_ABSTRACT	MDAS_integer	
MDAS_DOC	MDAS_integer	

MDAS_FORMAT attribute	Type	Use
(Input Lineage)		Formats share these attributes with other entities.
MDAS_FMT_IN	MDAS_integer	
MDAS_FMT_INC	MDAS_integer	
MDAS_FMT_INV	MDAS_integer array	

MDAS_FORMAT attribute	Type	Use
(Consequential Lineage)		Formats share these attributes with other entities.
MDAS_FMT_OUT	MDAS_integer	
MDAS_FMT_OUTC	MDAS_integer	
MDAS_FMT_OUTV	MDAS_integer array	

MDAS_FORMAT attribute	Type	Use
(Generation Lineage)		Formats share these attributes with other entities.
MDAS_FMT_GEN	MDAS_integer	
MDAS_FMT_GNP	MDAS_string	
MDAS_FMT_GNR	MDAS_integer	
MDAS_FMT_GNU	MDAS_integer	

MDAS_FORMAT attribute	Type	Use
(Re-Generation Policy)		Formats share these attributes with other entities.
MDAS_FMT_PLCY	MDAS_integer	

MDAS_FORMAT attribute	Type	Use
(Trigger)		Formats share these attributes with other entities.
MDAS_FMT_TRGM	MDAS_integer	
MDAS_FMT_TRGP	MDAS_integer	
MDAS_FMT_TRGC	MDAS_integer	
MDAS_FMT_TRGMV	MDAS_integer array	
MDAS_FMT_TRGPV	MDAS_integer array	

MDAS_FORMAT attribute	Type	Use
(Instantiation Date)		Formats share these attributes with other entities.
MDAS_FMT_DATE	MDAS_time	

MDAS_FORMAT attribute	Type	Use
(Instantiation Permanence)		Formats share these attributes with other entities.
MDAS_FMT_PERM	MDAS_double	
MDAS_FMT_PURG	MDAS_logical	

MDAS_FORMAT attribute	Type	Use
(Catalog SpecHist)		Formats share these attributes with other entities.
MDAS_FMT_HSA	MDAS_integer	
MDAS_FMT_HST	MDAS_time	
MDAS_FMT_HSS	MDAS_spectrum	
MDAS_FMT_HSC	MDAS_integer	
MDAS_FMT_HSAV	MDAS_integer array	
MDAS_FMT_HSTV	MDAS_time array	
MDAS_FMT_HSSV	MDAS_spectrum array	

MDAS_FORMAT attribute	Type	Use
(Catalog Perf)		Formats share these attributes with other entities.
MDAS_FMT_PERF	TBD	TBD.

4.7.2.3.2 MDAS_SERVER

A *server* is an MDAS_METHOD specialized for access to specific resources or storage mediums. For data sets, this might be the O/S for the storage resource or a specialized service provider such as a DBMS. The MDAS library is compiled with *drivers* for MDAS servers. The library matches the names of compiled drivers with server names (or id#'s) at run-time to determine access requirements for services requested a user.

Unique Method Attributes

None. A server is a method with MDAS_SERV_METH set to true.

Shared Attributes

An MDAS_SERVER has all the attributes of an MDAS_METHOD.

4.7.2.3.3 MDAS_POLICY

...

4.7.2.3.4 MDAS_ACTION

...

4.7.2.4 Collective Entities

4.7.2.4.1 MDAS_CD

A CD is an MDAS Catalog, or "Catalog Data". Catalogs are a data set with a specific use in MDAS.

Open question: are catalogs replicated?

Catalog metadata have some attributes common to data sets and methods, but are not partitioned into segments. Here, all attributes of catalogs are listed for completeness. Descriptions are only given for those attributes unique to catalogs.

Unique Catalog Attributes

TBD.

Shared Attributes

MDAS_CD attribute	Type	Use
(Base)		Catalogs share these attributes with other entities.
MDAS_NAME	MDAS_string	
MDAS_ID	MDAS_integer	

MDAS_CD attribute	Type	Use
(Version)		Catalogs share these attributes with other entities.
MDAS_VERSION	MDAS_string	
MDAS_VERSIONX	MDAS_logical	
MDAS_VERSIONM	MDAS_integer	
MDAS_VERSIONP	MDAS_integer	
MDAS_VERSIONN	MDAS_integer	

MDAS_CD attribute	Type	Use
(Alias)		Catalogs share these attributes with other entities.
MDAS_ALIAS	MDAS_string	
MDAS_ALIASC	MDAS_integer	
MDAS_ALIASV	MDAS_string array	

MDAS_CD attribute	Type	Use
(Documentation)		Catalogs share these attributes with other entities.
MDAS_ABSTRACT	MDAS_integer	
MDAS_DOC	MDAS_integer	

MDAS_CD attribute	Type	Use
(SD)		Catalogs share these attributes with other entities.
MDAS_SD_TABLE	MDAS_integer	
MDAS_SD_KEYC	MDAS_integer	
MDAS_SD_COLS	MDAS_string array	
MDAS_SD_KEY_TYP	MDAS_integer	
MDAS_SD_KEYS	MDAS_handle	
MDAS_SD_LOB_COL	MDAS_string	

MDAS_CD attribute	Type	Use
(Logical Group)		Catalogs share these attributes with other entities.
MDAS_STOR_GRPN	MDAS_string	
MDAS_STOR_GRPI	MDAS_integer	
MDAS_STOR_GRPNC	MDAS_integer	
MDAS_STOR_GRPNV	MDAS_string array	
MDAS_STOR_GRPIC	MDAS_integer	
MDAS_STOR_GRPIV	MDAS_integer array	

MDAS_CD attribute	Type	Use
(Logical Domain)		Catalogs share these attributes with other entities.
MDAS_STOR_DMNN	MDAS_string	
MDAS_STOR_DMNI	MDAS_integer	
MDAS_STOR_DMNNC	MDAS_integer	
MDAS_STOR_DMNNV	MDAS_string array	
MDAS_STOR_DMNIC	MDAS_integer	
MDAS_STOR_DMNIV	MDAS_integer array	

MDAS_CD attribute	Type	Use
(Input Lineage)		Catalogs share these attributes with other entities.
MDAS_STOR_IN	MDAS_integer	
MDAS_STOR_INC	MDAS_integer	
MDAS_STOR_INV	MDAS_integer array	

MDAS_CD attribute	Type	Use
(Consequential Lineage)		Catalogs share these attributes with other entities.
MDAS_STOR_OUT	MDAS_integer	
MDAS_STOR_OUTC	MDAS_integer	
MDAS_STOR_OUTV	MDAS_integer array	

MDAS_CD attribute	Type	Use
(Generation Lineage)		Catalogs share these attributes with other entities.
MDAS_STOR_GEN	MDAS_integer	
MDAS_STOR_GNP	MDAS_string	
MDAS_STOR_GNR	MDAS_integer	
MDAS_STOR_GNU	MDAS_integer	

MDAS_CD attribute	Type	Use
(Re-Generation Policy)		Catalogs share these attributes with other entities.
MDAS_STOR_PLCY	MDAS_integer	

MDAS_CD attribute	Type	Use
(Trigger)		Catalogs share these attributes with other entities.
MDAS_STOR_TRGM	MDAS_integer	
MDAS_STOR_TRGP	MDAS_integer	
MDAS_STOR_TRGC	MDAS_integer	
MDAS_STOR_TRGMV	MDAS_integer array	
MDAS_STOR_TRGPV	MDAS_integer array	

MDAS_CD attribute	Type	Use
(Storage Date)		Catalogs share these attributes with other entities.
MDAS_STOR_DATE	MDAS_time	

MDAS_CD attribute	Type	Use
(Storage Permanence)		Catalogs share these attributes with other entities.
MDAS_STOR_PERM	MDAS_double	
MDAS_STOR_PURG	MDAS_logical	

MDAS_CD attribute	Type	Use
(Storage Format)		Catalogs share these attributes with other entities.
MDAS_STOR_FMTN	MDAS_string	
MDAS_STOR_FMTI	MDAS_integer	

MDAS_CD attribute	Type	Use
(Storage Resource)		Catalogs share these attributes with other entities.
MDAS_STOR_RSCN	MDAS_string	
MDAS_STOR_RSCI	MDAS_integer	

MDAS_CD attribute	Type	Use
(Storage Server)		Catalogs share these attributes with other entities.
MDAS_STOR_SRVN	MDAS_string	
MDAS_STOR_SRVI	MDAS_integer	

MDAS_CD attribute	Type	Use
(Storage Path and Name)		Catalogs share these attributes with other entities.
MDAS_STOR_DIR	MDAS_string	
MDAS_STOR_NAM	MDAS_string	

MDAS_CD attribute	Type	Use
(Catalog Owner)		Catalogs share these attributes with other entities.
MDAS_STOR_OWN	MDAS_integer	

MDAS_CD attribute	Type	Use
(Catalog SpecHist)		Catalogs share these attributes with other entities.
MDAS_STOR_HSA	MDAS_integer	
MDAS_STOR_HST	MDAS_time	
MDAS_STOR_HSS	MDAS_spectrum	
MDAS_STOR_HSC	MDAS_integer	
MDAS_STOR_HSAV	MDAS_integer array	
MDAS_STOR_HSTV	MDAS_time array	
MDAS_STOR_HSSV	MDAS_spectrum array	

MDAS_CD attribute	Type	Use
(Catalog Perf)		Catalogs share these attributes with other entities.
MDAS_STOR_PERF	TBD	TBD.

MDAS_CD attribute	Type	Use
(Catalog Lock)		Catalogs share these attributes with other entities.
MDAS_STOR_RLCK	MDAS_logical	
MDAS_STOR_RLCKS	MDAS_time	
MDAS_STOR_RLCKE	MDAS_time	
MDAS_STOR_RLCKD	MDAS_integer	
MDAS_STOR_WLCK	MDAS_logical	
MDAS_STOR_WLCKS	MDAS_time	
MDAS_STOR_WLCKE	MDAS_time	
MDAS_STOR_WLCKD	MDAS_integer	

MDAS_CD attribute	Type	Use
(Catalog Security)		Catalogs share these attributes with other entities.
MDAS_STOR_AUTK	MDAS_string	
MDAS_STOR_AUTM	MDAS_integer	
MDAS_STOR_RACCK	MDAS_string	
MDAS_STOR_RACCM	MDAS_integer	
MDAS_STOR_WACCK	MDAS_string	
MDAS_STOR_WACCM	MDAS_integer	
MDAS_STOR_CRYK	MDAS_string	
MDAS_STOR_CRYM	MDAS_integer	

MDAS_CD attribute	Type	Use
(Replicates)		Catalogs share these attributes with other entities.
MDAS_REPLICATES	MDAS_integer	

MDAS_CD attribute	Type	Use
(Replicate Group)		Catalogs share these attributes with other entities.
MDAS_REPL_GRPN	MDAS_string	
MDAS_REPL_GRPNC	MDAS_integer array	
MDAS_REPL_GRPNV	MDAS_string array	
MDAS_REPL_GRPI	MDAS_integer	
MDAS_REPL_GRPIC	MDAS_integer array	
MDAS_REPL_GRPIV	MDAS_string array	

MDAS_CD attribute	Type	Use
(Replicate Domain)		Catalogs share these attributes with other entities.
MDAS_REPL_DMNN	MDAS_string	
MDAS_REPL_DMNNC	MDAS_integer array	
MDAS_REPL_DMNNV	MDAS_string array	
MDAS_REPL_DMNI	MDAS_integer	
MDAS_REPL_DMNIC	MDAS_integer array	
MDAS_REPL_DMNIV	MDAS_string array	

MDAS_CD attribute	Type	Use
(Repl. Input Lineage)		Catalogs share these attributes with other entities.
MDAS_REPL_IN	MDAS_integer	
MDAS_REPL_INC	MDAS_integer	
MDAS_REPL_INV	MDAS_integer array	

MDAS_CD attribute	Type	Use
(Repl. Conseq. Lineage)		Catalogs share these attributes with other entities.
MDAS_REPL_OUT	MDAS_integer	
MDAS_REPL_OUTVC	MDAS_integer	
MDAS_REPL_OUTV	MDAS_integer array	

MDAS_CD attribute	Type	Use
(Repl. Gen. Lineage)		Catalogs share these attributes with other entities.
MDAS_REPL_GEN	MDAS_integer	
MDAS_REPL_GNP	MDAS_string	
MDAS_REPL_GNR	MDAS_integer	
MDAS_REPL_GNU	MDAS_integer	

MDAS_CD attribute	Type	Use
(Replication Trigger)		Catalogs share these attributes with other entities.
MDAS_REPL_TRGM	MDAS_integer	
MDAS_REPL_TRGP	MDAS_integer	
MDAS_REPL_TRGC	MDAS_integer	
MDAS_REPL_TRGMV	MDAS_integer array	
MDAS_REPL_TRGPV	MDAS_integer array	

MDAS_CD attribute	Type	Use
(Repl. Re-Gen. Policy)		Catalogs share these attributes with other entities.
MDAS_REPL_PLCY	MDAS_integer	

MDAS_CD attribute	Type	Use
(Replicate Dates)		Catalogs share these attributes with other entities.
MDAS_REPL_DATE	MDAS_time	
MDAS_REPL_DATEV	MDAS_time array	

MDAS_CD attribute	Type	Use
(Replicate Permanence)		Catalogs share these attributes with other entities.
MDAS_REPL_PERM	MDAS_double	
MDAS_REPL_PERMV	MDAS_double array	
MDAS_REPL_PURG	MDAS_logical	
MDAS_REPL_PURGV	MDAS_logical array	

MDAS_CD attribute	Type	Use
(Replicate Format)		Catalogs share these attributes with other entities.
MDAS_REPL_FMTH	MDAS_logical	
MDAS_REPL_FMTN	MDAS_string	
MDAS_REPL_FMTNV	MDAS_string array	
MDAS_REPL_FMTI	MDAS_integer	
MDAS_REPL_FMTIV	MDAS_integer array	

MDAS_CD attribute	Type	Use
(Repl. Storage Resource)		Catalogs share these attributes with other entities.
MDAS_REPL_RSCH	MDAS_logical	
MDAS_REPL_RSCN	MDAS_string	
MDAS_REPL_RSCNV	MDAS_string array	
MDAS_REPL_RSCI	MDAS_integer	
MDAS_REPL_RSCIV	MDAS_integer array	

MDAS_CD attribute	Type	Use
(Repl. Storage Server)		Catalogs share these attributes with other entities.
MDAS_REPL_SRVH	MDAS_logical	
MDAS_REPL_SRVN	MDAS_string	
MDAS_REPL_SRVNV	MDAS_string array	
MDAS_REPL_SRVI	MDAS_integer	
MDAS_REPL_SRVIV	MDAS_integer array	

MDAS_CD attribute	Type	Use
(Repl. Dir. and Name)		Catalogs share these attributes with other entities.
MDAS_REPL_DIRN	MDAS_string	
MDAS_REPL_DIRNV	MDAS_string array	
MDAS_REPL_NAMN	MDAS_string	
MDAS_REPL_NAMNV	MDAS_string array	

MDAS_CD attribute	Type	Use
(Replicate Owner)		Catalogs share these attributes with other entities.
MDAS_REPL_OWN	MDAS_string	
MDAS_REPL_OWNV	MDAS_string array	

MDAS_CD attribute	Type	Use
(Replicate SpecHist)		Catalogs share these attributes with other entities.
MDAS_REPL_HSA	MDAS_integer	
MDAS_REPL_HST	MDAS_time	
MDAS_REPL_HSS	MDAS_spectrum	
MDAS_REPL_HSC	MDAS_integer	
MDAS_REPL_HSAV	MDAS_integer array	
MDAS_REPL_HSTV	MDAS_time array	
MDAS_REPL_HSSV	MDAS_spectrum array	

MDAS_CD attribute	Type	Use
(Replicate Lock)		Catalogs share these attributes with other entities.
MDAS_REPL_RLCK	MDAS_logical	
MDAS_REPL_RLCKS	MDAS_time	
MDAS_REPL_RLCKE	MDAS_time	
MDAS_REPL_RLCKD	MDAS_integer	
MDAS_REPL_WLCK	MDAS_logical	
MDAS_REPL_WLCKS	MDAS_time	
MDAS_REPL_WLCKE	MDAS_time	
MDAS_REPL_WLCKD	MDAS_integer	

MDAS_CD attribute	Type	Use
(Replicate Security)		Catalogs share these attributes with other entities.
MDAS_REPL_AUTK	MDAS_string	
MDAS_REPL_AUTM	MDAS_integer	
MDAS_REPL_RACCK	MDAS_string	
MDAS_REPL_RACCM	MDAS_integer	
MDAS_REPL_WACCK	MDAS_string	
MDAS_REPL_WACCM	MDAS_integer	
MDAS_REPL_CRYK	MDAS_string	
MDAS_REPL_CRYM	MDAS_integer	

MDAS_CD attribute	Type	Use
(Replicate Perf)		Catalogs share these attributes with other entities.
MDAS_REPL_PERF	TBD	TBD.

4.7.2.4.2 MDAS_DOMAIN

Domains are functionally equivalent to `MDAS_GROUP`, but have the additional constraint of access control.

Every user is a (singleton) domain, but may also have memberships in larger domains.

Domain Attributes

MDAS_DOMAIN attribute	Type	Use
(Base)		Domains share these attributes with other entities.
MDAS_NAME	MDAS_string	
MDAS_ID	MDAS_integer	
MDAS_CD_ID	MDAS_integer	

MDAS_DOMAIN attribute	Type	Use
(Version)		Domains share these attributes with other entities.
MDAS_VERSION	MDAS_string	
MDAS_VERSIONX	MDAS_logical	
MDAS_VERSIONM	MDAS_integer	
MDAS_VERSIONP	MDAS_integer	
MDAS_VERSIONN	MDAS_integer	

MDAS_DOMAIN attribute	Type	Use
(Alias)		Domains share these attributes with other entities.
MDAS_ALIAS	MDAS_string	
MDAS_ALIASC	MDAS_integer	
MDAS_ALIASV	MDAS_string array	

MDAS_DOMAIN attribute	Type	Use
(Documentation)		Domains share these attributes with other entities.
MDAS_ABSTRACT	MDAS_integer	
MDAS_DOC	MDAS_integer	

MDAS_DOMAIN attribute	Type	Use
(Input Lineage)		Domains share these attributes with other entities.
MDAS_DMN_IN	MDAS_integer	
MDAS_DMN_INC	MDAS_integer	
MDAS_DMN_INV	MDAS_integer array	

MDAS_DOMAIN attribute	Type	Use
(Consequential Lineage)		Domains share these attributes with other entities.
MDAS_DMN_OUT	MDAS_integer	
MDAS_DMN_OUTC	MDAS_integer	
MDAS_DMN_OUTV	MDAS_integer array	

MDAS_DOMAIN attribute	Type	Use
(Generation Lineage)		Domains share these attributes with other entities.
MDAS_DMN_GEN	MDAS_integer	
MDAS_DMN_GNP	MDAS_string	
MDAS_DMN_GNR	MDAS_integer	
MDAS_DMN_GNU	MDAS_integer	

MDAS_DOMAIN attribute	Type	Use
(Re-Generation Policy)		Domains share these attributes with other entities.
MDAS_DMN_PLCY	MDAS_integer	

MDAS_DOMAIN attribute	Type	Use
(Trigger)		Domains share these attributes with other entities.
MDAS_DMN_TRGM	MDAS_integer	
MDAS_DMN_TRGP	MDAS_integer	
MDAS_DMN_TRGC	MDAS_integer	
MDAS_DMN_TRGMV	MDAS_integer array	
MDAS_DMN_TRGPV	MDAS_integer array	

MDAS_DOMAIN attribute	Type	Use
(Storage Date)		Domains share these attributes with other entities.
MDAS_DMN_DATE	MDAS_time	

MDAS_DOMAIN attribute	Type	Use
(Storage Permanence)		Domains share these attributes with other entities.
MDAS_DMN_PERM	MDAS_double	
MDAS_DMN_PURG	MDAS_logical	

MDAS_DOMAIN attribute	Type	Use
(Domain Owner)		Domains share these attributes with other entities.
MDAS_DMN_OWN	MDAS_integer	

MDAS_DOMAIN attribute	Type	Use
(Domain SpecHist)		Domains share these attributes with other entities.
MDAS_DMN_HSA	MDAS_integer	
MDAS_DMN_HST	MDAS_time	
MDAS_DMN_HSS	MDAS_spectrum	
MDAS_DMN_HSC	MDAS_integer	
MDAS_DMN_HSAV	MDAS_integer array	
MDAS_DMN_HSTV	MDAS_time array	
MDAS_DMN_HSSV	MDAS_spectrum array	

MDAS_DOMAIN attribute	Type	Use
(Domain Perf)		Domains share these attributes with other entities.
MDAS_DMN_PERF	TBD	TBD.

MDAS_DOMAIN attribute	Type	Use
(Domain Security)		Domains share these attributes with other entities.
MDAS_DMN_AUTK	MDAS_string	
MDAS_DMN_AUTM	MDAS_integer	
MDAS_DMN_RACCK	MDAS_string	
MDAS_DMN_RACCM	MDAS_integer	
MDAS_DMN_WACCK	MDAS_string	
MDAS_DMN_WACCM	MDAS_integer	
MDAS_DMN_EACCK	MDAS_string	
MDAS_DMN_EACCM	MDAS_integer	
MDAS_DMN_CRYK	MDAS_string	
MDAS_DMN_CRYM	MDAS_integer	

4.7.2.4.3 MDAS_GROUP

A *group* is a set of MDAS entities with a common attribute encapsulated in the group name.

Group Attributes

MDAS_GROUP attribute	Type	Use
(Base)		Groups share these attributes with other entities.
MDAS_NAME	MDAS_string	
MDAS_ID	MDAS_integer	
MDAS_CD_ID	MDAS_integer	

MDAS_GROUP attribute	Type	Use
(Version)		Groups share these attributes with other entities.
MDAS_VERSION	MDAS_string	
MDAS_VERSIONX	MDAS_logical	
MDAS_VERSIONM	MDAS_integer	
MDAS_VERSIONP	MDAS_integer	
MDAS_VERSIONN	MDAS_integer	

MDAS_GROUP attribute	Type	Use
(Alias)		Groups share these attributes with other entities.
MDAS_ALIAS	MDAS_string	
MDAS_ALIASC	MDAS_integer	
MDAS_ALIASV	MDAS_string array	

MDAS_GROUP attribute	Type	Use
(Documentation)		Groups share these attributes with other entities.
MDAS_ABSTRACT	MDAS_integer	
MDAS_DOC	MDAS_integer	

MDAS_GROUP attribute	Type	Use
(Input Lineage)		Groups share these attributes with other entities.
MDAS_GRP_IN	MDAS_integer	
MDAS_GRP_INC	MDAS_integer	
MDAS_GRP_INV	MDAS_integer array	

MDAS_GROUP attribute	Type	Use
(Consequential Lineage)		Groups share these attributes with other entities.
MDAS_GRP_OUT	MDAS_integer	
MDAS_GRP_OUTC	MDAS_integer	
MDAS_GRP_OUTV	MDAS_integer array	

MDAS_GROUP attribute	Type	Use
(Generation Lineage)		Groups share these attributes with other entities.
MDAS_GRP_GEN	MDAS_integer	
MDAS_GRP_GNP	MDAS_string	
MDAS_GRP_GNR	MDAS_integer	
MDAS_GRP_GNU	MDAS_integer	

MDAS_GROUP attribute	Type	Use
(Re-Generation Policy)		Groups share these attributes with other entities.
MDAS_GRP_PLCY	MDAS_integer	

MDAS_GROUP attribute	Type	Use
(Trigger)		Groups share these attributes with other entities.
MDAS_GRP_TRGM	MDAS_integer	
MDAS_GRP_TRGP	MDAS_integer	
MDAS_GRP_TRGC	MDAS_integer	
MDAS_GRP_TRGMV	MDAS_integer array	
MDAS_GRP_TRGPV	MDAS_integer array	

MDAS_GROUP attribute	Type	Use
(Storage Date)		Groups share these attributes with other entities.
MDAS_GRP_DATE	MDAS_time	

MDAS_GROUP attribute	Type	Use
(Storage Permanence)		Groups share these attributes with other entities.
MDAS_GRP_PERM	MDAS_double	
MDAS_GRP_PURG	MDAS_logical	

MDAS_GROUP attribute	Type	Use
(Domain Owner)		Groups share these attributes with other entities.
MDAS_GRP_OWN	MDAS_integer	

MDAS_GROUP attribute	Type	Use
(Domain SpecHist)		Groups share these attributes with other entities.
MDAS_GRP_HSA	MDAS_integer	
MDAS_GRP_HST	MDAS_time	
MDAS_GRP_HSS	MDAS_spectrum	
MDAS_GRP_HSC	MDAS_integer	
MDAS_GRP_HSAV	MDAS_integer array	
MDAS_GRP_HSTV	MDAS_time array	
MDAS_GRP_HSSV	MDAS_spectrum array	

MDAS_GROUP attribute	Type	Use
(Domain Perf)		Groups share these attributes with other entities.
MDAS_GRP_PERF	TBD	TBD.

4.7.2.4.4 MDAS_LIST

An entity list is a simple list of entity records. It typically encountered in the `result` of `MDAS_INQUIRE()`.

An `MDAS_LIST` is created with

```
MDAS_INFO_CREATE( MDAS_LIST, listinfo, status )
```

where `listinfo` is an `infoh` structure initialized by the call.

To add `Info` structure `myinfo` to a list, use

```
MDAS_INFO_ADD_LATTR( listinfo, myinfo, n, status ).
```

The list element number n assigned to the entity is returned by the call. To retrieve the nth `Info` structure from a list, use

```
MDAS_INFO_SCAN_LATTR( listinfo, $n$, someinfo, status )
```

where `someinfo` is an `infoh` structure (re)initialized by the call. List entities can be deleted with

```
MDAS_INFO_DEL_LATTR( listinfo, $n$, status ).
```

The number of entities in a list is given by attribute `MDAS_LIST_COUNT` and the types of each entity is given by `MDAS_LIST_TYPE`. These attributes may be scanned with `MDAS_INFO_SCAN_ATTR()`.

To create "list entities" without explicitly declaring individual entity handles, use

```
MDAS_INFO_CREATE_LEATTR( entitytype, listinfo, $n$, status )
```

where `listinfo` is the name of an existing `MDAS_LIST Info` structure and `entitytype` is a valid MDAS entity. The list element number n assigned to the entity is returned by the call.

To set attributes of "list entities" without the explicit entity handles, use

```
MDAS_INFO_SET_LEATTR( attr, value, listinfo, $n$, status )
```

where `attr` is a valid attribute for the entity type and `value` is a valid value for the attribute. To read values directly from a list entity, use

```
MDAS_INFO_SCAN_LEATTR( listinfo, $n$, attr, value, status )
```

List entity replicate (LER) values can likewise be accessed with

```
MDAS_INFO_SET_LERATTR( $r$, attr, value, listinfo, $n$, status )
```

and

```
MDAS_INFO_SCAN_LERATTR( listinfo, $n$, $r$, attr, value, status )
```

where r is the replicate index of the desired attribute.

List Attributes

MDAS_METHOD attribute	Type	Use
MDAS_LIST_COUNT	MDAS_integer	Number of entities in list.
MDAS_LIST_TYPEV	MDAS_integer array	Array of entity types corresponding to each entity in the list.
MDAS_LIST_INFOV	MDAS_infoh array	Array of Info handles.

4.7.2.4.5 MDAS_SITE

A *site* is an `MDAS_RESOURCE` that identifies a set of resources known to users as a "site". The `MDAS_SITE` token has special meaning to the MDAS Library, but is otherwise equivalant to a resource entity. Individual resources of a site are registered as segments of the site entity.

Unique Method Attributes

None. A site is a resource with `MDAS_SITE_RSRC` set to true.

Shared Attributes

An `MDAS_SITE` has all the attributes of an `MDAS_RESOURCE`.

4.7.2.5 Directive Entities

The MDAS Library defines several entities for specifying directives in `Info` to MDAS Library routines and directive flags as attributes of Catalog metadata.

4.7.2.5.1 MDAS_DIRECTIVE

`MDAS_DIRECTIVE` is functionally equivalent to `MDAS_LIST`, but its use is restricted to MDAS Library calls requiring argument lists placed in an `Info` structure.

All attributes of MDAS_DIRECTIVE are optional. The semantics of any particular attribute is dependent upon its use. Most members are designed as flags.

MDAS_DIRECTIVE
    (zero or more of:)
        MDAS_INPUT
        MDAS_OUTPUT
        MDAS_READ
        MDAS_WRITE
        MDAS_READWRITE
        MDAS_APPEND
        MDAS_SPOOL
        ...

4.7.2.6 Conditional Entities

Conditional entities are used wherever an Info structure must specify logical semantics to an MDAS Library function. A primary use is the cond argument to MDAS_INQUIRE() (section 4.7.3.2.1).

For example, suppose a user is interested in finding Catalog id#'s for any entity whose name equals "kilroy". To specify this query to MDAS_INQUIRE(), a user program would place the following attribute in the info argument

MDAS_ENTITY	MDAS_ANY	(null)

and these two attributes in the cond argument:

MDAS_CRITERIA	MDAS_NAME	"kilroy"
MDAS_LIMIT	MDAS_ANY	MDAS_ID

Upon successful return from MDAS_INQUIRE(), the calling program would receive a list in the result parameter containing Catalog id#'s for entities matching the query:

MDAS_LIST
    MDAS_LIST_COUNT *n*
    ...

More information on Conditional Entities is given in the following subsections.

4.7.2.6.1 MDAS_ANY

MDAS_ANY is a wild-card *token* which is used in the context of queries.

4.7.2.6.2 MDAS_CONDITION

MDAS_CONDITION is used to specify logical conditions to MDAS Library functions. These conditions may be listed sequentially, or built in tree-like hierarchies to specify complex logic requirements. Like most conditional entities, all attributes of MDAS_CONDITION are optional.

MDAS_CONDITION
- MDAS_EQUIV
- MDAS_NOT
- MDAS_AND
- MDAS_OR
- MDAS_NAND
- MDAS_NOR
- MDAS_ANY

4.7.2.6.3 MDAS_CRITERIA

MDAS_CRITERIA
- MDAS_TOKEN
- MDAS_SUBSTR
- MDAS_LIKE
- MDAS_ORIGIN
- MDAS_BEST
- MDAS_DISTINCT
- MDAS_ANY

4.7.2.6.4 MDAS_LIMIT

MDAS_LIMIT
- (any valid entity)
  - (any attribute of the entity)
- MDAS_ANY

4.7.2.6.5 MDAS_EXTENT

MDAS_EXTENT
- MDAS_LOCAL
- MDAS_WIDE
- MDAS_UNIVERSE
- MDAS_ANY

4.7.3 API Prototypes

The following descriptions of MDAS function prototypes are provided as a reference to the MDAS High-Level API. Example use of each function is given in procedural pseudo-code. For a tutorial style introduction, please see the *MDAS User Guide and Tutorial.*

4.7.3.1 Library Initialization and Cleanup

The MDAS Library requires initialization of internal parameters and run-time resources before use. Likewise, a clean-up process is required when the application program no longer needs MDAS library resources. The routines `MDAS_INIT()` and `MDAS_FINALIZE()` are provided for this purpose.

4.7.3.1.0.1 MDAS_INIT()

```
MDAS_INIT(argc, argv, comm, status)
     argv:   (IN)      array of character string
     argc:   (IN)      integer
     comm:   (IN)      handle
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	MDAS library initialized
MDAS_WARN_MDASRC	warning	no run-time mdasrc data found
MDAS_WARN_TICKETS	warning	no run-time ticket data found
MDAS_WARN_MPI	warning	comm non-NULL but no MPI
MDAS_ERR_MEMORY	error	unable to allocate memory

`MDAS_INIT()` allocates memory for basic library operations and initializes library run-time parameters as described in section 4.5. The arguments `argv` and `argc` are the standard string and count structures for command line argument lists encountered in Unix and other operating systems. The `comm` argument is used only when the application is linked to an MPI-enabled version of MDAS; it is otherwise ignored.

4.7.3.1.0.2 MDAS_FINALIZE()

```
MDAS_FINALIZE(comm,status)
     comm:   (IN)      handle
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	MDAS library finalized
MDAS_WARN_COMM	warning	comm different from MDAS_INIT
MDAS_WARN_MPI	warning	comm non-NULL but no MPI
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_MEMORY	error	unable to free all allocations
MDAS_ERR_CONNECT	error	unable to close all connections
MDAS_ERR_COMM	error	comm not part of MDAS_INIT comm

MDAS_FINALIZE() resets library parameters, closes any open connections, and frees any memory allocated by MDAS library calls.

4.7.3.1.0.3 MDAS_INIT() and MDAS_FINALIZE() Examples

The following example assumes that argv and argc are system-defined variables.

```
PROGRAM dolittle
     ...
     MDAS_status status
     ...
     MDAS_INIT(argc, argv, NULL, status)
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM
```

The following example assumes that argv and argc are system-defined variables. The handle MPI_COMM_WORLD is defined by MPI.

```
PROGRAM do_mpi
     ...
     integer ierr
     MDAS_status status
     ...
     MPI_INIT(ierr)
     ...
     MDAS_INIT(argc, argv, MPI_COMM_WORLD, status)
     ...
     MDAS_FINALIZE(MPI_COMM_WORLD,status)
     ...
     MPI_FINALIZE(ierr)
     ...
END PROGRAM
```

4.7.3.2 Info Operations on MDAS Catalogs

4.7.3.2.1 Obtaining Info From MDAS Catalogs

MDAS_INFO_INQUIRE() operates on catalogs of MDAS metadata stored in external metadata servers. The existence and location of such servers are identified either at (a) MDAS installation time and/or (b) application run-time. See sections 4.5 and 4.13 for details.

4.7.3.2.1.1 MDAS_INFO_INQUIRE()

```
MDAS_INFO_INQUIRE(info, cond, extent, result, status)
     info:   (IN)      MDAS_INFOH
     cond:   (IN)      MDAS_INFOH
     extent: (IN)      MDAS_INFOH
     result: (OUT)     MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	result(s) obtained
MDAS_WARN_INFO	warning	info is empty or NULL
MDAS_WARN_COND	warning	cond is empty or NULL
MDAS_WARN_EXTENT	warning	extent is empty or NULL
MDAS_WARN_RESULT	warning	result is empty
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_INFO	error	invalid info
MDAS_ERR_COND	error	invalid cond
MDAS_ERR_EXTENT	error	invalid extent
MDAS_ERR_RESULT	error	invalid result
MDAS_ERR_MEMORY	error	unable to allocate memory

The info argument should specify known static information about the desired item, however sparse or incomplete. For example, a user may wish to specify the MDAS_TYPE and a portion of the name or descriptive keywords in info.

The cond argument is used to specify *conditions* and *criteria* binding on the query. The tokens MDAS_CONDITION and MDAS_CRITERIA are provided for this purpose. (See section 4.7.2.6.)

The extent of catalogs searched are specified in extent. MDAS_EXTENT may be used in combination with MDAS_CONDITION and extent to fully qualify the catalog search boundaries. If NULL is given by the user, default catalogs specified by the run-time environment are searched.

Query results are returned in result (which may be internally spooled). If no matches are found, result is returned empty and an error is returned in status. Users are re-

sponsible for freeing the memory allocated for **result** with either MDAS_INFO_FREE() or MDAS_FINALIZE().

4.7.3.2.1.2 MDAS_INFO_INQUIRE() Example

4.7.3.2.2 Catalog Info Registration

MDAS_INFO_REGISTER() operates on catalogs of MDAS metadata stored in external metadata servers. The existence and location of such servers are identified either at MDAS installation time (section 4.13) and/or application run-time (section 4.5).

To register **Info** about users, data sets, methods, or resources to an MDAS Catalog, a minimal set of attributes is required. These may be specified by the user, and in some cases are automatically generated by the MDAS Library.

4.7.3.2.2.1 MDAS_INFO_REGISTER()

```
MDAS_INFO_REGISTER(info, extent, status)
     info:   (IN/OUT)  MDAS_INFOH
     extent: (IN)      MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	**info** registered
MDAS_WARN_INFO	warning	**info** is empty
MDAS_WARN_EXTENT	warning	**extent** is empty or NULL
MDAS_WARN_REGISTER	warning	**info** previously registered
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_INFO	error	invalid **info**
MDAS_ERR_EXTENT	error	invalid **extent**
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_REGISTER	error	insufficient metadata

MDAS_INFO_REGISTER() adds information about users, data sets, methods, and resources to one or more catalogs. Specifying NULL in **extent** will select the default catalog. If **info** does not contain the minimal metadata required for registration in an MDAS Catalog, an error is returned in **status** and attributes (with empty values) of the required attributes are appended to **info**.

4.7.3.2.2.2 MDAS_INFO_REGISTER() Example

```
PROGRAM reginfo
     ...
```

```
      MDAS_status  status
      MDAS_INFOH   myinfo
      ...
      MDAS_INIT(argc, argv, NULL, status)
      ...
      MDAS_INFO_CREATE(MDAS_METHOD, myinfo, status)
      MDAS_INFO_SET_ATTR(MDAS_NAME, "myfilter", myinfo, status)
      MDAS_INFO_SET_ATTR(MDAS_STOR_RSCN, "moon.com", myinfo, status)
      MDAS_INFO_SET_ATTR(MDAS_STOR_DIR, "/users/kilroy/bin/", myinfo, status)
      ...
      MDAS_INFO_REGISTER(myinfo, NULL, status)
      ...
      MDAS_FINALIZE(NULL,status)
      ...
END PROGRAM
```

4.7.3.3 Returning Status Messages

The MDAS Library provides 3 procedures to generate status messages from `status` codes. All of these procedures take `status` vectors as input. The contents of `status` is not altered.

4.7.3.3.0.3 MDAS_STATUS_PRINT()

```
      MDAS_STATUS_PRINT(status)
           status: (IN)      MDAS_status
```

`MDAS_STATUS_PRINT` is one of a very few routines that are guaranteed to work outside of `MDAS_INIT()` and `MDAS_FINALIZE()`. It interprets the contents of `status` and prints the corresponding status message(s) to the O/S equivalent of standard output, one line per message. `MDAS_STATUS_PRINT` ignores `status(0)` and `status(1)`.

4.7.3.3.0.4 MDAS_STATUS_MSG()

```
      MDAS_STATUS_MSG(code, procid, msg)
           code:   (IN)      integer
           procid: (IN)      integer
           msg:    (OUT)     MDAS_string
```

`MDAS_STATUS_MSG` is also guaranteed to work outside of `MDAS_INIT()` and `MDAS_FINALIZE()`. The `code` argument should contain one MDAS status bit code corresponding to MDAS Library procedure id# `procid`. If `code` and `procid` are valid, then up to 31 characters of string data are returned in `msg`. Otherwise, `msg` is empty on return.

4.7.3.3.0.5 MDAS_STATUS_INFO()

```
MDAS_STATUS_INFO(status, info)
     status: (IN)      MDAS_status
     info:   (OUT)     MDAS_INFOH
```

MDAS_STATUS_INFO requires that the MDAS Library be initialized. It creates an Info structure in info from the contents of status. No structure is created if status is invalid. MDAS_STATUS_INFO ignores status(0) and status(1).

4.7.3.3.0.6 Example use of MDAS_STATUS_PRINT() and MDAS_STATUS_MSG()

In the following examples, program checkstat examines status, prints any error or warning messages, then exits on error. Program printstat prints the entire table of MDAS status codes. Program checkrc checks to see if MDAS_INIT() found resource information in the local environment. If not, the status message associated with this condition is printed and the program exits.

```
PROGRAM checkstat
     ...
     MDAS_status  status
     ...
     MDAS_INIT(argc, argv, NULL, status)
     if (status(0) .ne. 0)
          MDAS_STATUS_PRINT(status)
          if (status(0) < 0) exit
     end if
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM

PROGRAM printstat
     ...
     MDAS_status  status
     integer      p
     ...
     status(2) = FFFFFFFF
     for status(3) = 1, MDAS_PROC_COUNT
          MDAS_STATUS_PRINT(status)
     end for
     ...
END PROGRAM

PROGRAM checkrc
```

```
     ...
     MDAS_status  status
     character    msg(32)
     ...
     MDAS_INIT(argc, argv, NULL, status)
     if (status(2) & MDAS_WARN_MDASRC)
          MDAS_STATUS_MSG(MDAS_WARN_MDASRC, status(3), msg)
          MDAS_FINALIZE(NULL,status)
          print msg
          exit
     end if
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM
```

4.7.3.4 Info Management

4.7.3.4.1 Creating Your Own Info

MDAS Info structures can be created by either initializing a new MDAS_INFOH with MDAS_INFO_CREATE() or copying an existing Info structure into a new area of memory with MDAS_INFO_DUP(). Existing Info structures can be purged from program memory with MDAS_INFO_FREE().

4.7.3.4.1.1 MDAS_INFO_CREATE()

```
MDAS_INFO_CREATE(entity, info, status)
     entity: (IN)      MDAS_token
     info:   (OUT)     MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	info created
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_INFO	error	invalid info
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_TOKEN	error	invalid MDAS_token

MDAS_INFO_CREATE() allocates memory for an MDAS_info structure for the entity specified by entity. No calls are made to MDAS Catalogs. To place values in the structure, use MDAS_INFO_SET_ATTR(), MDAS_INFO_SET_RATTR() (for attributes of replicates), MDAS_INFO_ADD_LATTR() (for attributes of lists), MDAS_INFO_SET_LEATTR() (for list entities), and MDAS_INFO_SET_LERATTR() (for list entity replicates).

4.7.3.4.1.2 MDAS_INFO_DUP()

```
MDAS_INFO_DUP(oldinfo, newinfo, status)
     info1:  (IN)      MDAS_INFOH
     info2:  (OUT)     MDAS_INFOH
```

status codes	type	meaning
MDAS_SUCCESS	success	**info2** created from **info1**
MDAS_WARN_INFO	warning	**info1** is empty
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_MEMORY	error	unable to allocate memory

MDAS_INFO_DUP() allocates memory for the contents of **info2** and then initializes it with the contents of **info1**.

4.7.3.4.1.3 MDAS_INFO_FREE()

```
MDAS_INFO_FREE(info, status)
     info:   (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	**info** deallocated
MDAS_WARN_INFO	warning	**info** is empty
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_INFO	error	invalid **info**
MDAS_ERR_MEMORY	error	unable to free all allocations

MDAS_INFO_FREE() deallocates all memory associated with **info**. Upon return, the value of **info** is NULL. Any Info structures not explicitly freed with MDAS_INFO_FREE() will be purged during MDAS_FINALIZE().

4.7.3.4.1.4 Example use of MDAS_INFO_CREATE() and MDAS_INFO_FREE()

```
PROGRAM createinfo
     ...
     MDAS_status  status
     MDAS_INFOH   dsinfo
     ...
     MDAS_INIT(argc, argv, NULL, status)
     ...
     MDAS_INFO_CREATE(MDAS_DATASET, dsinfo, status)
     ...
```

```
     MDAS_INFO_FREE(dsinfo, status)
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM
```

4.7.3.4.1.5 MDAS_INFO_DUP() Example

```
PROGRAM dupinfo
     ...
     MDAS_status  status
     MDAS_INFOH   dsinfo1, dsinfo2
     ...
     MDAS_INIT(argc, argv, NULL, status)
     ...
     MDAS_INFO_CREATE(MDAS_DATASET, dsinfo1, status)
     ...
     MDAS_INFO_DUP(dsinfo1, dsinfo2, status)
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM
```

4.7.3.4.2 Setting Info Attributes

4.7.3.4.2.1 MDAS_INFO_SET_ATTR()

MDAS_INFO_SET_ATTR() sets a metadata attribute in an existing MDAS_info structure.

```
     MDAS_INFO_SET_ATTR(attr, value, info, status)
          attr:   (IN)      MDAS_token
          value:  (IN)      pointer (choice)
          info:   (IN/OUT)  MDAS_INFOH
          status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	added 1 attribute to info
MDAS_WARN_NULL	warning	added NULL attribute
MDAS_WARN_TOKEN	warning	NULL MDAS_token
MDAS_WARN_VALUE	warning	NULL value
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_TOKEN	error	invalid MDAS_token
MDAS_ERR_DATATYPE	error	invalid MDAS_datatype for value
MDAS_ERR_INFO	error	invalid info

4.7.3.4.2.2 MDAS_INFO_SET_ATTR() Example

```
PROGRAM addinfo
     ...
     MDAS_status  status
     MDAS_INFOH   dsinfo
     ...
     MDAS_INIT(argc, argv, NULL, status)
     ...
     MDAS_INFO_CREATE(MDAS_DATASET, dsinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_NAME, "fin.dat", dsinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_NAME, "netcdf", dsinfo, status)
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM
```

4.7.3.4.3 Scanning Info Attributes

4.7.3.4.3.1 MDAS_INFO_SCAN_ATTR()

```
MDAS_INFO_SCAN_ATTR(info, attr, value, status)
     info:   (IN)      MDAS_INFOH
     attr:   (OUT)     MDAS_token
     value:  (OUT)     pointer (choice)
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	item(s) scanned
MDAS_WARN_ATTR	warning	attribute value is NULL
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_INFO	error	invalid info
MDAS_ERR_ATTR	error	invalid attribute for entity

MDAS_INFO_SCAN_ATTR() returns the attr attribute value for a given Info entity. A warning is returned in status if info has no attributes.

4.7.3.4.4 MDAS_LIST Entities and Attributes

An entity list is a simple list of entity records. It typically encountered in the result of MDAS_INQUIRE().

An MDAS_LIST is created with

```
MDAS_INFO_CREATE(MDAS_LIST, ilist, status)
```

where ilist is an infoh structure initialized by the call.

The number of entities in a list is given by attribute MDAS_LIST_COUNT and the types of each entity is given by MDAS_LIST_TYPE. These attributes may be scanned with MDAS_INFO_SCAN_ATTR().

4.7.3.4.4.1 MDAS_INFO_ADD_LATTR()

```
MDAS_INFO_ADD_LATTR(ilist, myinfo, n, status)
     ilist:  (IN/OUT)  MDAS_INFOH
     myinfo: (IN/OUT)  MDAS_INFOH
     n:      (OUT)     MDAS_integer
     status: (IN/OUT)  MDAS_status
```

Use MDAS_INFO_ADD_LATTR to add myinfo to list ilist. The list element number n assigned to the entity is returned by the call.

4.7.3.4.4.2 MDAS_INFO_SCAN_LATTR()

To retrieve the nth Info structure from a list, use

```
MDAS_INFO_SCAN_LATTR(ilist, n, myinfo, status)
     ilist:  (IN/OUT)  MDAS_INFOH
     n:      (IN)      MDAS_integer
     myinfo: (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

where someinfo is an infoh structure (re)initialized by the call.

4.7.3.4.4.3 MDAS_INFO_DEL_LATTR()

```
MDAS_INFO_DEL_LATTR(ilist, n, status)
     ilist:  (IN/OUT)  MDAS_INFOH
     n:      (IN)      MDAS_integer
     status: (IN/OUT)  MDAS_status
```

MDAS_INFO_DEL_LATTR() deletes list entities.

4.7.3.4.4.4 MDAS_INFO_CREATE_LEATTR()

```
MDAS_INFO_CREATE_LEATTR(type, ilist, n, status)
     type:   (IN)      MDAS_token
     ilist:  (IN/OUT)  MDAS_INFOH
     n:      (OUT)     MDAS_integer
     status: (IN/OUT)  MDAS_status
```

To create "list entities" without explicitly declaring individual entity handles, use MDAS_INFO_CREATE_LEATTR(where ilist is the name of an existing MDAS_LIST Info structure and type is a valid MDAS entity type. The list element number n assigned to the entity is returned by the call.

4.7.3.4.4.5 MDAS_INFO_SET_LEATTR()

```
MDAS_INFO_SET_LEATTR(attr, value, ilist, n, status)
     attr:   (IN)      MDAS_token
     value:  (IN)      pointer (choice)
     ilist:  (IN/OUT)  MDAS_INFOH
     n:      (IN)      MDAS_integer
     status: (IN/OUT)  MDAS_status
```

Use MDAS_INFO_SET_LEATTR() to set attributes of "list entities" without the explicit entity handles. The token attr must be a valid attribute for the entity type and value must be a valid value for the attribute.

4.7.3.4.4.6 MDAS_INFO_SCAN_LEATTR()

To read values directly from a list entity, use:

```
MDAS_INFO_SCAN_LEATTR(ilist, n, attr, value, status)
     ilist:  (IN/OUT)  MDAS_INFOH
     n:      (IN)      MDAS_integer
     attr:   (IN)      MDAS_token
     value:  (IN)      pointer (choice)
     status: (IN/OUT)  MDAS_status
```

4.7.3.4.4.7 MDAS_INFO_SET_LERATTR()

```
MDAS_INFO_SET_LERATTR(r, attr, value, ilist, n, status)
     r:      (IN)      MDAS_integer
     attr:   (IN)      MDAS_token
     value:  (IN)      pointer (choice)
     ilist:  (IN/OUT)  MDAS_INFOH
     n:      (IN)      MDAS_integer
     status: (IN/OUT)  MDAS_status
```

List entity replicate (LER) values can be accessed with MDAS_INFO_SET_LERATTR() where r is the replicate index of the desired attribute.

4.7.3.4.4.8 MDAS_INFO_SCAN_LERATTR()

```
MDAS_INFO_SCAN_LERATTR(ilist, n, r, attr, value, status)
     ilist:  (IN/OUT)  MDAS_INFOH
     n:      (IN)      MDAS_integer
     r:      (IN)      MDAS_integer
     attr:   (IN)      MDAS_token
     value:  (IN)      pointer (choice)
     status: (IN/OUT)  MDAS_status
```

4.7.3.4.5 Selecting Info

Queries on MDAS Catalogs by `MDAS_INQUIRE()` will often return metadata on multiple items in `result`. The user can of course manually transverse the `MDAS_LIST`, then (based on some user-defined decision mechanism) copy a particular item to a new `Info` structure for further use. Another alternative is to rely on default selection criteria when supplying several items in a single `Info` structure to a procedure that must use exactly one; e.g., `MDAS_CONNECT()`, `MDAS_OPEN()`, `MDAS_PIPE()`, etc.

A third option is to explicitly narrow the *result* to a single item according to some condition or criteria. `MDAS_INFO_SELECT()` is provided for this purpose. The use of this function is not limited to metadata returned by `MDAS_INQUIRE()`. The user may wish to input `Info` created at the user level or obtained by other means.

4.7.3.4.5.1 MDAS_INFO_SELECT()

```
MDAS_INFO_SELECT(info, cond, entity, status)
     info:   (IN/OUT)  MDAS_INFOH
     cond:   (IN)      MDAS_INFOH
     entity: (OUT)     MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	entity selected
MDAS_WARN_INFO	warning	`info` is empty or `NULL`
MDAS_WARN_COND	warning	`cond` is empty or `NULL`
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_INFO	error	invalid `info`
MDAS_ERR_COND	error	invalid `cond`

`MDAS_INFO_SELECT()` selects entities from `info` based on the criteria given in `cond` and returns a single item in `entity`. Only the contents of `info` are used: MDAS Catalogs are not searched.

The cond argument is used to specify *conditions* and *criteria* binding on the query. The tokens MDAS_CONDITION and MDAS_CRITERIA are provided for this purpose (see section 4.7.2.6). The cond argument may be NULL to force default selection rules.

If cond contains MDAS_BEST, then all records of info are searched and default selection rules apply. Otherwise, MDAS_INFO_SELECT() returns the first entity in info matching all the conditions and criteria in cond.

4.7.3.4.6 Printing Info

4.7.3.4.6.1 MDAS_INFO_PRINT()

```
MDAS_INFO_PRINT(info, status)
     info:   (IN)      MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	info printed
MDAS_WARN_INFO	warning	info is empty
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_INFO	error	invalid info

MDAS_INFO_PRINT() prints the contents of an Info structure to standard output or the O/S equivalent.

4.7.3.5 Access

4.7.3.5.1 Access Authorization

Applications running in a heterogeneous, distributed massive data analysis system may often require connections to resources, methods, and data sets which belong to different security realms or domains. Thus, the system must handle issues related to authentication and security in this environment. The *ticket* model is used within MDAS. Functionally, this model subsumes other security schemes.

The semantics of tickets is based on a point-to-point authorization protocol in which each side is assumed to have a private security key for the other, plus the ability to generate public "tickets" which can be decoded by the other party's private key for access authorization. Under this model, a passwordless system can be considered to have null tickets. A password challenge system can permit the instantiation of tickets, where the user changes from a login+password paradigm to a *ticket + login* paradigm. Given the permission to execute a method to access a data set and then to execute an analysis or display method, MDAS supports third-party authentication in which the methods directly exchange tickets to validate the information exchange.

4.7.3.5.1.1 MDAS_TICKET()

```
MDAS_TICKET(item, user, type, ticket, status)
     item:   (IN)      MDAS_INFOH
     user:   (IN)      MDAS_INFOH
     ttype:  (IN)      MDAS_INFOH
     ticket: (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	**ticket** updated
MDAS_WARN_USER	warning	**user** is empty or NULL
MDAS_WARN_TTYPE	warning	**ttype** is empty or NULL
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_USER	error	invalid or unknown **user**
MDAS_ERR_TTYPE	error	invalid or unknown **ttype**
MDAS_ERR_TICKET	error	invalid **ticket**
MDAS_ERR_MEMORY	error	unable to allocate memory

MDAS_TICKET() generates an authorization "ticket" of type **ttype** for a specific **item** and **user** using built-in libraries or modules installed on the local platform. The **item** can be a data set, method, or resource. The **user** and **ttype** arguments may reference an actual user and authorization method, or NULL may be supplied to force default behavior. Any metadata referenced by **item**, **user**, and **ttype** must be registered in an accessible MDAS Catalog. If the descriptions of **item**, **user**, or **ttype** are somehow incomplete, MDAS_INQUIRE() and other **Info** manipulation procedures may be called internally with default conditions and criteria to complete the minimal metadata requirements for generating **ticket**. MDAS_TICKET fails if the supplied metadata is too vague or local methods are not found to enact the requested authentication.

4.7.3.5.2 Service Connection

4.7.3.5.2.1 MDAS_CONNECT()

```
MDAS_CONNECT(server, ticket, comm, servh, status)
     server: (IN/OUT)  MDAS_INFOH
     ticket: (IN/OUT)  MDAS_INFOH
     comm:   (IN)      handle
     servh:  (OUT)     MDAS_SERVH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	connection established
MDAS_WARN_SERVER	warning	server is empty or NULL
MDAS_WARN_TICKET	warning	ticket is empty or NULL
MDAS_WARN_COMM	warning	comm different from MDAS_INIT
MDAS_WARN_MPI	warning	comm non-NULL but no MPI
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_SERVER	error	invalid or unknown server
MDAS_ERR_TICKET	error	invalid or unknown ticket
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_COMM	error	comm not part of MDAS_INIT comm
MDAS_ERR_SERVH	error	server not available

MDAS_CONNECT() connects an application program with the authorization protocol and key specified in ticket to the service described in server. If server contains an MDAS_LIST describing multiple services then a single service will be selected using default selection rules. The ticket argument may also contain multiple tickets, in which case a match between one of the tickets and a selected service will be attempted. The ticket argument may be NULL to force internal ticket development with default selection criteria. If the descriptions of server or ticket are somehow incomplete, MDAS_INQUIRE() and other Info manipulation procedures may be called internally with default conditions and criteria to complete the minimal metadata requirements for generating the connection.

The comm argument should be used only when the application is linked to an MPI-enabled version of MDAS. If non-NULL, comm must be equivalent to or "split" (MPI_COMM_SPLIT()) from the communicator argument used in MDAS_INIT().

The returned servh handle maintains the functionality of a physical connection across time-outs.

4.7.3.5.2.2 MDAS_DISCONNECT()

```
MDAS_DISCONNECT(servh, comm, status)
     servh:  (IN/OUT)  MDAS_SERVH
     comm:   (IN)      MDAS_handle
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	server deallocated
MDAS_WARN_COMM	warning	comm different from MDAS_CONNECT
MDAS_WARN_MPI	warning	comm non-NULL but no MPI
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_SERVH	error	invalid servh
MDAS_ERR_MEMORY	error	unable to deallocate memory
MDAS_ERR_COMM	error	comm not part of MDAS_CONNECT
MDAS_ERR_SERVER	error	server not available

MDAS_DISCONNECT() closes the connection specified by **servh** (and possibly **comm**) and frees any memory associated with the structure.

4.7.3.5.2.3 MDAS_CONNECT and MDAS_DISCONNECT() Example

```
PROGRAM dbconnect
     ...
     MDAS_status  status
     MDAS_INFOH   dbinfo
     MDAS_SERVH   dbh
     ...
     MDAS_INIT(argc, argv, NULL, status)
     ...
     MDAS_INFO_CREATE(MDAS_RESOURCE, dbinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_NAME, "ord.com", dbinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_RSRC_SRVN, "illustra", dbinfo, status)
     MDAS_CONNECT(dbinfo, NULL, NULL, servh, status)
     ...
     MDAS_DISCONNECT(servh, NULL, status)
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM
```

4.7.3.6 Operations on Data Sets

4.7.3.6.1 Cache Operations

4.7.3.6.1.1 MDAS_GET()

```
MDAS_GET(dset, cache, status)
     dset:   (IN/OUT)  MDAS_INFOH
     cache:  (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	dset cached
MDAS_WARN_CACHE	warning	cache is empty or NULL
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DSET	error	invalid dset
MDAS_ERR_CACHE	error	invalid cache
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_READ	error	read access denied
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_IO	error	unable to perform I/O

MDAS_GET() retrieves a data set specified in dset and caches it on the local file system. Any required *tickets* and/or *connections* are automatically generated. The target location is selected by the MDAS library and returned in cache. The user can specify alternate formats for the storage of dset in cache on input. The MDAS library may invoke a 3rd party mover to perform the data transfer and possibly spool the data. If cache contains a named data set, then the operation of MDAS_GET() will be functionally equivalent to MDAS_INFO_PIPE(). If the description of dset is somehow incomplete, MDAS_INQUIRE() and other Info manipulation procedures may be called internally with default conditions and criteria to complete the minimal metadata requirements for identifying the data set.

4.7.3.6.1.2 MDAS_GETALL()

```
MDAS_GETALL(dset, cache, status)
     dset:   (IN/OUT)  MDAS_INFOH
     cache:  (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	data sets cached
MDAS_WARN_CACHE	warning	cache is empty or NULL
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DSET	error	invalid dset
MDAS_ERR_CACHE	error	invalid cache
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_READ	error	read access denied
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_IO	error	unable to perform I/O

MDAS_GETALL() retrieves a set of data sets specified in dset and caches them on the local file system. The entity type of dset may be either MDAS_DATASET or MDAS_LIST. The cache argument may not specify more than one resource and may not name any data set(s). MDAS_GETALL() is otherwise equivalent to MDAS_GET().

4.7.3.6.1.3 MDAS_PUT()

```
MDAS_PUT(dset, rsrc, status)
     dset:   (IN/OUT)  MDAS_INFOH
     rsrc:   (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	dset stored in rsrc
MDAS_WARN_RESOURCE	warning	rsrc is empty or NULL
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DSET	error	invalid dset
MDAS_ERR_RESOURCE	error	invalid rsrc
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_READ	error	read access denied
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_IO	error	unable to perform I/O

MDAS_PUT() retrieves a data set specified in dset from local cache and moves it to the resource specified by rsrc. The target location is selected by the MDAS library and returned in rsrc. The user can also use rsrc to specify alternate formats and storage servers for the storage of dset. The MDAS library may invoke a 3rd party mover to perform the data transfer and possibly spool the data. If rsrc contains a named data set, then the operation of MDAS_PUT() will be functionally equivalent to MDAS_INFO_PIPE(). If the descriptions of dset or rsrc are somehow incomplete, MDAS_INQUIRE() and other Info manipulation procedures may be called internally with default conditions and criteria to complete the minimal metadata requirements for identifying the data set and resource.

4.7.3.6.1.4 MDAS_PUTALL()

```
MDAS_PUTALL(dset, rsrc, status)
     dset:   (IN/OUT)  MDAS_INFOH
     rsrc:   (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	dset stored in rsrc
MDAS_WARN_RESOURCE	warning	rsrc is empty or NULL
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DSET	error	invalid dset
MDAS_ERR_RESOURCE	error	invalid rsrc
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_READ	error	read access denied
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_IO	error	unable to perform I/O

MDAS_PUTALL() retrieves a set of data sets specified in dset from local cache and moves them to the resource specified by rsrc. The entity type of dset may be either MDAS_DATASET or MDAS_LIST. The rsrc argument may not specify more than one resource and may not name any data set(s). MDAS_PUTALL() is otherwise equivalent to MDAS_PUT().

4.7.3.6.1.5 MDAS_GET() Example

```
PROGRAM getk2
     ...
     MDAS_status  status
     MDAS_INFOH   dsinfo, cacheinfo
     ...
     MDAS_INIT(argc, argv, NULL, status)
     ...
     MDAS_INFO_CREATE(MDAS_DATASET, dsinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_NAME, "fin.dat", dsinfo, status)
     MDAS_INFO_CREATE(MDAS_DATASET, cacheinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_STOR_FMTN, "khoros2", cacheinfo, status)
     MDAS_GET(dsinfo, cacheinfo, status)
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM
```

4.7.3.6.2 Generalized Point-to-Point Dataflow

MDAS provides a generalization of the Unix "pipe" paradigm for data set transimission. The operation may be performed on either open data handles or data sets described by metadata in Info structures. Since MDAS_INFO_PIPE() allows the MDAS Library (and any schedulers known or discovered by the library) to select resources, tickets, and methods for completing the pipe transaction, it provides the greatest opportunities for MDAS optimization of any point-to-point data transfer request.

4.7.3.6.2.1 MDAS_INFO_PIPE()

```
MDAS_INFO_PIPE(info1, info2, status)
     info1:  (IN/OUT)  MDAS_INFOH
     info2:  (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	pipe completed
MDAS_WARN_READ	warning	source automatically selected
MDAS_WARN_WRITE	warning	target automatically selected
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_WARN_FORMAT	warning	format conversion performed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_INFO1	error	invalid **info1**
MDAS_ERR_INFO2	error	invalid **info2**
MDAS_ERR_READ	error	read access denied
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to perform I/O
MDAS_ERR_FORMAT	error	unable to convert format

MDAS_INFO_PIPE() takes two MDAS **MDAS_info** structures—each containing references to data sets, and "pipes" the contents of a single data set from the first **MDAS_info** structure to a single data set in the second. If **info1** contains records describing multiple data sets then a single data set will be selected using default selection rules. The **info2** argument may also contain multiple data sets, in which case a single data set will be selected. Any required *tickets* and/or *connections* are automatically generated.

If the descriptions of the (selected) data sets in **info1** or **info2** are somehow incomplete, MDAS_INQUIRE() and other **Info** manipulation procedures may be called internally with default conditions and criteria to complete the minimal metadata requirements for completing the "pipe" transaction. The call fails when choices are ambiguous. The call is otherwise functionally equivalent to MDAS_DATAH_PIPE().

4.7.3.6.2.2 MDAS_DATAH_PIPE()

```
MDAS_DATAH_PIPE(dh1, dh2, status)
     dh1:    (IN/OUT)  MDAS_DATAH
     dh2:    (IN/OUT)  MDAS_DATAH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	stream dh1 piped to stream dh2
MDAS_WARN_EOF	warning	dh1 already at EOF
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_WARN_FORMAT	warning	format conversion performed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DH1	error	invalid dh1
MDAS_ERR_DH2	error	invalid dh2
MDAS_ERR_READ	error	dh1 closed
MDAS_ERR_WRITE	error	dh2 closed
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to perform I/O
MDAS_ERR_FORMAT	error	unable to convert format

MDAS_DATAH_PIPE() takes two MDAS data handles—one open for read and the other for write, and "pipes" the data from one to the other performing any stream and format conversion in-situ. Third-party movers are employed if necessary. The operation begins at the current stream locations in dh1 and dh2. The call may fail if no movers are available to perform the required operation(s). Both handles are open on successful return but assumed to be at "end of stream".

4.7.3.6.2.3 MDAS_INFO_PIPE() Example

```
PROGRAM pipefun
     ...
     MDAS_status  status
     MDAS_INFOH   funinfo, psinfo
     ...
     MDAS_INIT(argc, argv, NULL, status)
     ...
     MDAS_INFO_CREATE(MDAS_DATASET, funinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_NAME, "fun.dat", funinfo, status)
     ...
     MDAS_INFO_CREATE(MDAS_RESOURCE, psinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_RSRC_TYPE, MDAS_PRINTER, psinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_RSRC_FMTN, "postscript", psinfo, status)
     ...
     MDAS_INFO_PIPE(funinfo, psinfo, status)
     print "fun.dat printed at:"
     MDAS_INFO_PRINT(psinfo,status)
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM
```

4.7.3.6.2.4 MDAS_DATAH_PIPE() Example

```
PROGRAM pipestream
     ...
     MDAS_status  status
     MDAS_INFOH   fininfo, tailinfo
     MDAS_DATAH   finh, tailh
     ...
     MDAS_INIT(argc, argv, NULL, status)
     ...
     MDAS_INFO_CREATE(MDAS_DATASET, fininfo, status)
     MDAS_INFO_SET_ATTR(MDAS_NAME, "fin.dat", fininfo, status)
     MDAS_OPEN(NULL, NULL, fininfo, NULL, finh, status)
     ...
     MDAS_INFO_CREATE(MDAS_DATASET, tailinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_NAME, "tail.dat", tailinfo, status)
     MDAS_OPEN(NULL, NULL, tailinfo, NULL, finh, status)
     ...
     MDAS_DATAH_PIPE(finh, tailh, status)
     ...
     MDAS_CLOSE(finh, NULL, status)
     MDAS_CLOSE(tailh, NULL, status)
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM
```

4.7.3.6.3 Generalized Data Set Scatter/Gather

MDAS supplies multiplex versions of the MDAS_*_PIPE() commands for the replication and concatenation of multiple data sets.

4.7.3.6.3.1 MDAS_INFO_SCATTER()

```
MDAS_INFO_SCATTER(info1, info2, status)
     info1:  (IN/OUT)  MDAS_INFOH
     info2:  (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	scatter completed
MDAS_WARN_READ	warning	source automatically selected
MDAS_WARN_WRITE	warning	target automatically selected
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_WARN_FORMAT	warning	format conversion performed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_INFO1	error	invalid info1
MDAS_ERR_INFO2	error	invalid info2
MDAS_ERR_READ	error	read access denied
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to perform I/O
MDAS_ERR_FORMAT	error	unable to convert format

MDAS_INFO_SCATTER() replicates the contents of a single data set specified in **info1** to one or more data sets referenced in **info2**. The result is that the contents of a data set in **info1** are replicated to one or more data sets in **info2**. If **info1** contains records describing multiple data sets then a single data set will be selected using default selection rules. If **info2** contains multiple data sets, all will be selected as targets. Any required *tickets* and/or *connections* are automatically generated.

If the descriptions of the (selected) data sets in **info1** or **info2** are somehow incomplete, MDAS_INQUIRE() and other Info manipulation procedures may be called internally with default conditions and criteria to complete the minimal metadata requirements for completing the "scatter" transaction. The call fails when choices are ambiguous, access authorization is denied, or when proper read/write modes can not be established. The call is otherwise functionally equivalent to MDAS_DATAH_SCATTER().

4.7.3.6.3.2 MDAS_INFO_GATHER()

```
MDAS_INFO_GATHER(info1, info2, status)
     info1:  (IN/OUT)  MDAS_INFOH
     info2:  (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	gather completed
MDAS_WARN_READ	warning	source automatically selected
MDAS_WARN_WRITE	warning	target automatically selected
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_WARN_FORMAT	warning	format conversion performed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_INFO1	error	invalid **info1**
MDAS_ERR_INFO2	error	invalid **info2**
MDAS_ERR_READ	error	read access denied
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to perform I/O
MDAS_ERR_FORMAT	error	unable to convert format

MDAS_INFO_GATHER() pipes the contents of one or more data sets referenced in **info1** to a single data set specified in **info2**. The result is that the contents of the data sets in **info1** are concatenated (in list order) to the data set in **info2**. If **info1** contains multiple data sets, all will be selected as sources. If **info2** contains records describing multiple data sets then a single data set will be selected using default selection rules. Any required *tickets* and/or *connections* are automatically generated.

If the descriptions of the (selected) data sets in **info1** or **info2** are somehow incomplete, MDAS_INQUIRE() and other Info manipulation procedures may be called internally with default conditions and criteria to complete the minimal metadata requirements for completing the "gather" transaction. The call fails when choices are ambiguous, access authorization is denied, or when proper read/write modes can not be established. The call is otherwise functionally equivalent to MDAS_DATAH_GATHER().

4.7.3.6.3.3 MDAS_DATAH_SCATTER()

```
MDAS_DATAH_SCATTER(dh, count, dhvect, status)
     dh:     (IN/OUT)  MDAS_DATAH
     count:  (IN)      integer
     dhvect: (IN/OUT)  vector of MDAS_DATAH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	dh piped to dhvect
MDAS_WARN_EOF	warning	dh already at EOF
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_WARN_FORMAT	warning	format conversion performed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DH	error	invalid dh
MDAS_ERR_COUNT	error	invalid count
MDAS_ERR_DHVECT	error	invalid dhvect
MDAS_ERR_READ	error	dh closed
MDAS_ERR_WRITE	error	handle in dhvect closed
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to perform I/O
MDAS_ERR_FORMAT	error	unable to convert format

MDAS_DATAH_SCATTER() pipes the contents of the data stream pointed to by dh to a vector of distinct streams dhvect of size count. The operation begins at the current stream locations in dh and dhvect. The result is that the (remaining) stream contents of dh are replicated to the streams in dhvect. Upon successful return, all data handles will be open but assumed to be at "end of stream". Read mode must be enabled for dh and all data handles in dhvect must be enabled for write.

4.7.3.6.3.4 MDAS_DATAH_GATHER()

```
MDAS_DATAH_GATHER(count, dhvect, dh, status)
     count:  (IN)      integer
     dhvect: (IN/OUT)  vector of MDAS_DATAH
     dh:     (IN/OUT)  MDAS_DATAH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	dhvect concatenated to dh
MDAS_WARN_EOF	warning	some dhvect handles at EOF
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_WARN_FORMAT	warning	format conversion performed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_COUNT	error	invalid count
MDAS_ERR_DHVECT	error	invalid dhvect
MDAS_ERR_DH	error	invalid dh
MDAS_ERR_READ	error	handle in dhvect closed
MDAS_ERR_WRITE	error	dh closed
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to perform I/O
MDAS_ERR_FORMAT	error	unable to convert format

MDAS_DATAH_GATHER() pipes the contents of the distinct data streams pointed to by dhvect of size count to the stream dh. The operation begins at the current stream locations in dhvect and dh. The result is that the (remaining) stream contents of dhvect are concatenated to the stream in dh *in vector order*. Upon successful return, all data handles will be open but assumed to be at "end of stream". Read mode must be enabled for all data handles in dhvect and dh must be enabled for write.

4.7.3.6.4 Generalized Open on Data Sets

4.7.3.6.4.1 MDAS_OPEN()

```
MDAS_OPEN(servh, comm, dset, mode, dh, status)
     servh:   (IN)      MDAS_SERVH
     comm:    (IN)      MDAS_handle
     dset:    (IN)      MDAS_INFOH
     mode:    (IN)      MDAS_INFOH
     dh:      (OUT)     MDAS_DATAH
     status:  (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	data set open on dh
MDAS_WARN_SERVH	warning	servh is empty or NULL
MDAS_WARN_SERVER	warning	server automatically selected
MDAS_WARN_COMM	warning	comm different from MDAS_CONNECT
MDAS_WARN_MPI	warning	comm non-NULL but no MPI
MDAS_WARN_DSET	warning	dset is empty or NULL
MDAS_WARN_DATASET	warning	data set automatically selected
MDAS_WARN_MODE	warning	mode is empty or NULL
MDAS_WARN_IO	warning	I/O mode automatically selected
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_SERVH	error	invalid servh
MDAS_ERR_COMM	error	comm not part of MDAS_CONNECT
MDAS_ERR_DSET	error	invalid dset
MDAS_ERR_MODE	error	invalid mode
MDAS_ERR_READ	error	read access denied
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_SERVER	error	server not available
MDAS_ERR_DATASET	error	data set not available

MDAS_OPEN() opens a generalized data handle for a data set specified in the dset argument. If servh is NULL, then the metadata specified in dset are used to establish access authentication and a service connection to a resource containing the data set. The comm argument is used only when the application is linked to an MPI-enabled version of MDAS; it is otherwise ignored. If comm is used, it must be equivalent or "split" from the MPI

communicator used to create servh. This behavior is guaranteed when servh is NULL. The MDAS mode argument should be either NULL for default read/write access or contain an I/O directive attribute.

A warning is returned in status if NULL is passed but the data set access mode is limited to one of read or write. An error is returned with a NULL dh when access to the data set is denied. If the descriptions of servh, dset, or mode are somehow incomplete, MDAS_INQUIRE() and other Info manipulation procedures may be called internally with default conditions and criteria to complete the minimal metadata requirements for opening the data handle. If successful, the returned handle contains Info from dset and a pointer to the actual physical data stream employed.

4.7.3.6.4.2 MDAS_CLOSE()

```
MDAS_CLOSE(dh, comm, status)
     dh:     (IN/OUT)  MDAS_DATAH
     comm:   (IN)      MDAS_handle
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	dh deallocated
MDAS_WARN_COMM	warning	comm different from MDAS_OPEN
MDAS_WARN_MPI	warning	comm non-NULL but no MPI
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DH	error	invalid dh
MDAS_ERR_COMM	error	comm not part of MDAS_OPEN
MDAS_ERR_MEMORY	error	unable to deallocate memory
MDAS_ERR_SERVER	error	server not available
MDAS_ERR_IO	error	I/O error on close

MDAS_CLOSE() closes the data stream, etc. specified by dh (and possibly comm) and frees any memory associated with the structure.

4.7.3.6.4.3 Example use of MDAS_OPEN() and MDAS_CLOSE()

```
PROGRAM openfinh
     ...
     MDAS_status  status
     MDAS_INFOH   dsinfo
     MDAS_DATAH   finh
     ...
     MDAS_INIT(argc, argv, NULL, status)
     ...
     MDAS_INFO_CREATE(MDAS_DATASET, dsinfo, status)
```

```
     MDAS_INFO_SET_ATTR(MDAS_NAME, "fin.dat", dsinfo, status)
     MDAS_INFO_SET_ATTR(MDAS_STOR_FMT, "hdf", dsinfo, status)
     ...
     MDAS_OPEN(NULL, NULL, dsinfo, NULL, finh, status)
     ...
     MDAS_CLOSE(finh, NULL, status)
     ...
     MDAS_FINALIZE(NULL,status)
     ...
END PROGRAM
```

4.7.3.7 Block I/O

MDAS supplies `MDAS_READ()` and `MDAS_WRITE()` for block I/O on data handles. The development of spooler daemons are one example use.

```
MDAS_READ(dh, type, count, buffer, status)
     dh:     (IN/OUT)  MDAS_DATAH
     type:   (IN)      MDAS_datatype
     count:  (IN)      integer
     buffer: (OUT)     MDAS_handle (user defined buffer)
     status: (IN/OUT)  MDAS_status

MDAS_WRITE(type, count, buffer, dh, status)
     type:   (IN)      MDAS_datatype
     count:  (IN)      integer
     buffer: (IN)      MDAS_handle (user defined buffer)
     dh:     (IN/OUT)  MDAS_DATAH
     status: (IN/OUT)  MDAS_status
```

The argument lists to `MDAS_READ` and `MDAS_WRITE` differ only in the order of arguments. Valid `type` values are given in table 4.5. The `count` specifies how many of the given `type` are contiguously packed in the given `buffer`. All data supplied in `buffer` by `MDAS_READ()` will be *packed*. For `MDAS_WRITE()`, if `type` is an `MDAS_INFOH`, `MDAS_TABLE_HANDLE`, or `MDAS_INFOH`, a handle packed by `MDAS_HANDLE_PACK()` must be supplied in `buffer`.

4.7.3.7.1 Data Handle Conversion

MDAS provides several data handle conversion routines to permit native protocol transactions on data sets opened by `MDAS_OPEN()`. For example, it is desirable in data mining and knowledge building applications to search (inquire) about data sets available for access with a file stream interface and then perform operations on those data sets. Conversions *to* data handles are also permitted in the event that a user wishes to perform `MDAS_DATAH_PIPE()` on an "unregistered" data set. In particular: data received in a C program from standard input outside the purview of MDAS cannot be registered.

4.7.3.7.1.1 MDAS_CVT_DH_U()

```
MDAS_CVT_DH_U(dh, unit, status)
     dh:     (IN/OUT)  MDAS_DATAH
     unit:   (IN/OUT)  integer
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	unit assigned for dh
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DH	error	invalid dh
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to convert I/O stream
MDAS_ERR_UNIT	error	unable to assign to given unit

MDAS_CVT_DH_U() is most applicable to Fortran implementations. If the caller supplies unit < 1, then a unit number is selected and returned. The user should not call Fortran close() on unit. To close the stream, use MDAS_CLOSE() on the original dh.

4.7.3.7.1.2 MDAS_CVT_DH_FP()

```
MDAS_CVT_DH_FP(dh, fp, status)
     dh:     (IN/OUT)  MDAS_DATAH
     fp:     (OUT)     MDAS_handle
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	fp created for dh
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DH	error	invalid dh
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to convert I/O stream

MDAS_CVT_DH_FP() is most applicable to ANSI C implementations. The returned (FILE*) fp is also cached on dh. The user should not call C fclose() on fp. To close the stream, use MDAS_CLOSE() on the original dh.

4.7.3.7.1.3 MDAS_CVT_DH_FS()

```
MDAS_CVT_DH_FS(dh, fs, status)
     dh:     (IN/OUT)  MDAS_DATAH
```

```
fs:     (OUT)     MDAS_handle
status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	fs created for dh
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DH	error	invalid dh
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to convert I/O stream

MDAS_CVT_DH_FS() creates an open socket fs for the stream in dh. The user should not call manually close fs, but rather use MDAS_CLOSE() on the original dh.

4.7.3.7.1.4 MDAS_CVT_U_DH()

```
MDAS_CVT_U_DH(unit, dh, status)
     unit:   (IN)      integer
     dh:     (OUT)     MDAS_DATAH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	dh created for unit
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DH	error	invalid dh
MDAS_ERR_UNIT	error	invalid unit
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to convert I/O stream

MDAS_CVT_U_DH() is most applicable to Fortran implementations. The caller must supply an valid unit number to an open file or stream. To close the data handle and stream, the user should first call MDAS_CLOSE() on dh (which will only deallocate the data handle) and then call Fortran close() on on the original unit.

4.7.3.7.1.5 MDAS_CVT_FP_DH()

```
MDAS_CVT_FP_DH(fp, dh, status)
     fp:     (IN)      MDAS_handle
     dh:     (OUT)     MDAS_DATAH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	dh created for fp
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DH	error	invalid dh
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to convert I/O stream

MDAS_CVT_FP_DH() is most applicable to C and C++ implementations. The caller must supply an valid file handle fp to an open file or stream. To close the data handle and stream, the user should first call MDAS_CLOSE() on dh (which will only deallocate the data handle) and then call C fclose() on on the original fp.

4.7.3.7.1.6 MDAS_CVT_FS_DH()

```
MDAS_CVT_FS_DH(fs, dh, status)
     fs:     (IN)      MDAS_handle
     dh:     (OUT)     MDAS_DATAH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	dh created for fs
MDAS_WARN_IO	warning	3rd party mover employed
MDAS_ERR_INIT	error	MDAS not initialized
MDAS_ERR_DH	error	invalid dh
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_IO	error	unable to convert I/O stream

MDAS_CVT_DH_FP() creates a data handle for the open socket fs. The caller must supply an valid socket handle fs to an open stream. To close the data handle and stream. the user should first call MDAS_CLOSE() on dh (which will only deallocate the data handle) and then call close() on on the original fs.

4.7.3.7.1.7 MDAS_CVT_FP_DH() Example

```
/* PROGRAM cvtfp -- ANSI C */
     main(argc, argv) ;
     {
     ...
     FILE*        fp ;
     mdasC_status  status ;
     mdasC_INFOH   dsinfo ;
     mdasC_DATAH   finh, tailh ;
     ...
```

```
    fp = fopen("fin.dat", "r") ;
    ...
    mdasC_init(argc, argv, NULL, status) ;
    ...
    mdasC_cvt_fp_dh(fp, finh, status) ;
    ...
    mdasC_info_create(MDAS_DATASET, MDAS_NAME, "tail.dat", dsinfo, status) ;
    mdasC_open(NULL, NULL, dsinfo, NULL, tailh, status) ;
    ...
    mdasC_close(finh, NULL, status) ;
    fclose(finh) ;
    ...
    mdasC_close(tailh, NULL, status) ;
    ...
    mdasC_finalize(NULL,status)
    ...
    fclose(fp) ;
    ...
    exit(0) ;
    }
/* END PROGRAM */
```

4.7.3.8 Executing Methods

4.7.3.8.1 MDAS_EXEC()

```
MDAS_EXEC(method, params, rsrc, ds_in, ds_out, status)
     method: (IN/OUT)  MDAS_INFOH
     params: (IN/OUT)  MDAS_INFOH
     rsrc:   (IN/OUT)  MDAS_INFOH
     ds_in:  (IN/OUT)  MDAS_INFOH
     ds_out: (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	request completed
MDAS_WARN_METHOD	warning	method is empty or NULL
MDAS_WARN_PARAMS	warning	params is empty or NULL
MDAS_WARN_RESOURCE	warning	rsrc is empty or NULL
MDAS_WARN_INPUT	warning	ds_in is empty or NULL
MDAS_WARN_OUTPUT	warning	ds_out is empty or NULL
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_SERVER	error	server not available
MDAS_ERR_METHOD	error	method error

MDAS_EXEC discussion: TBD.

MDAS Data Type	Fortran 90 Type	ANSI C, C++ Type
`MDAS_table`	derived `TYPE`	`void**`
`MDAS_graph`	derived `TYPE`	`struct`

Table 4.5: MDAS Mid-Level data types and their counterparts in standard languages.

4.8 MDAS Mid-Level Interface

This section discusses the MDAS Mid-Level in detail. Datatypes exposed to the user at this level are discussed in section 4.8.1. Function prototypes are presented in section 4.8.2.

4.8.1 Mid-Level Data Types

In addition to the types supported at the API level (4.7.1), MDAS supports types for use at the Mid-Level library interface. A list of currently supported types is given in table ??. The implementation of these types is language and architecture dependent. Type conversions between languages and architectures is performed by Mid-Level routines in the MDAS library.

4.8.1.1 Graph

TBD.

The `MDAS_graph` structure is the primary protocol for communications between an MDAS-enabled application and an MDAS daemon. It is also used by higher level libraries for direct interface with the MDAS trasparency and transaction engines. Internally, an `MDAS_graph` structure is a directed graph whose nodes and vertices are defined by `MDAS_info` structures. An interface to `MDAS_graph` handles is given in section ??.

4.8.1.2 Table

TBD.

The MDAS Library defines the type `MDAS_table` for direct import and export of DBMS tables. The `MDAS_table` type most often appears as a `value` with token `MDAS_SD` in an `MDAS_info` structure and thus unseen by users. Input/Output operations on large tables may be internally spooled. An interface to `MDAS_table` handles is given in section ??.

4.8.1.3 Handle Structures

TBD.

MDAS Handle Type	Purpose
MDAS_GH	handle to MDAS_graph structure
MDAS_TABLH	handle to MDAS_table structure

Table 4.6: Additional MDAS handles.

Needs to describe details of handle structures.

The MDAS Mid-Level Library defines handles for MDAS_graph and MDAS_table structures, along with routines to manipulate the attributes of MDAS-defined handle types. This section discusses handle attributes and structures. A list of Mid-Level MDAS_handle types is given in table 4.6.

4.8.2 Mid-Level Prototypes

TBD.

4.8.2.1 Library Initialization

MDAS_INITIALIZED() allows layered libraries to determine if another library in the application code has called MDAS_INIT() or MDAS_FINALIZE(). It returns success (0) if the library is initialized. Otherwise, a warning code is returned indicating whether the library has (a) never been initialized or (b) has been finalized.

```
MDAS_INITIALIZED(status)
     status: (IN/OUT)  MDAS_status
```

4.8.2.2 Data Type Length and Packing

To support the import, export, and internal transfer of data among MDAS-enabled applications and external environments, the MDAS Library provides an interface to obtain the length of data structures in the system.

```
MDAS_DATATYPE_LEN(type, len, status)
     type:   (IN)      MDAS_datatype
     len:    (OUT)     integer
     status: (IN/OUT)  MDAS_status
```

MDAS_DATATYPE_LEN() returns the length (in bytes) of an MDAS_datatype (table 4.5, section ??) in the current run-time. If type is a handle, the handle length and not the length of the underlying structure is returned. Use MDAS_HANDLE_PACK_LEN() to obtain the contiguous length of an MDAS_handle.

```
MDAS_HANDLE_PACK_LEN(htype, handle, len, status)
     htype:  (IN)      Name of MDAS_handle
     handle: (IN/OUT)  MDAS_handle
     len:    (OUT)     integer
     status: (IN/OUT)  MDAS_status
```

MDAS_HANDLE_PACK_LEN() returns the length (in bytes) of a structure referenced by handle—as if it were packed into a contiguous stream of bytes. The routine may actually pack the structure if internal optimizations deem it prudent. Otherwise, the structure is left untouched. The value supplied in htype must (currently) be one of MDAS_GRAPH_HANDLE, MDAS_INFOH, or MDAS_TABLE_HANDLE.

```
MDAS_HANDLE_PACK(htype, handle, buffer, buflen, len, status)
     htype:  (IN)      Name of MDAS_handle
     handle: (IN/OUT)  MDAS_handle
     buffer: (IN/OUT)  MDAS_handle (user defined buffer)
     buflen: (IN)      integer
     len:    (OUT)     integer
     status: (IN/OUT)  MDAS_status
```

MDAS_HANDLE_PACK() *packs* a structure referenced by handle into a contiguous stream of bytes which is suitable for I/O. The buffer must be user-allocated memory of length buflen. The actual length of the packed structure is returned in len. This is guarenteed to be the same length as MDAS_HANDLE_PACK_LEN() would return for the same handle reference. A *packed* MDAS_handle structure is functionally equivalent to its unpacked form.

Note: subsequent operations on the content of a packed MDAS_handle structure (e.g., section ??) may cause the structure to be unpacked. When an MDAS_handle structure is packed for I/O or communication, its content should not be examined or updated until the I/O transaction is complete.

4.8.2.3 Program Graph Interface

One of the prime MDAS objectives is the replacement of the Unix file paradigm with a high-level data set abstraction. To do so also requires the abstraction of programs to methods, since programs are typically stored as executable *files*. Visual programming environments have proven to be an effective interface for the design and prototyping of software systems. The MDAS "Program Graph Interface" provides a direct means for visual programming GUIs to interface the MDAS transparency and transaction engines. MDAS Daemons and many High-Level MDAS API functions such as MDAS_GET(), MDAS_PUT(), MDAS_*_PIPE(), and MDAS_*_MPLX() are implemented with this Mid-Level interface.

```
MDAS_GRAPH_CREATE(gh, status)
     gh:     (IN/OUT)  MDAS_GRAPH_HANDLE
     status: (IN/OUT)  MDAS_status
```

```
MDAS_GRAPH_DUP(gh1, gh2, status)
     gh1:    (IN/OUT)  MDAS_GRAPH_HANDLE
     gh2:    (IN/OUT)  MDAS_GRAPH_HANDLE
     status: (IN/OUT)  MDAS_status


MDAS_GRAPH_ADD_NODE(info, gh, status)
     info:   (IN)      MDAS_INFOH
     gh:     (IN/OUT)  MDAS_GRAPH_HANDLE
     status: (IN/OUT)  MDAS_status


MDAS_GRAPH_DEL_NODE(info, gh, status)
     info:   (IN)      MDAS_INFOH
     gh:     (IN/OUT)  MDAS_GRAPH_HANDLE
     status: (IN/OUT)  MDAS_status


MDAS_GRAPH_CONNECT(info1, info2, cinfo, gh, status)
     info1:  (IN)      MDAS_INFOH
     info2:  (IN)      MDAS_INFOH
     cinfo:  (IN)      MDAS_INFOH
     gh:     (IN/OUT)  MDAS_GRAPH_HANDLE
     status: (IN/OUT)  MDAS_status


MDAS_GRAPH_CUT(info1, info2, gh, status)
     info1:  (IN)      MDAS_INFOH
     info2:  (IN)      MDAS_INFOH
     gh:     (IN/OUT)  MDAS_GRAPH_HANDLE
     status: (IN/OUT)  MDAS_status


MDAS_GRAPH_SELECT(info, gh, status)
     info:   (IN)      MDAS_INFOH
     gh:     (IN/OUT)  MDAS_GRAPH_HANDLE
     status: (IN/OUT)  MDAS_status


MDAS_GRAPH_CURSOR_SCAN(gh, action, scalar, status)
     gh:     (IN/OUT)  MDAS_GRAPH_HANDLE
     action: (IN)      integer
     scalar: (OUT)     integer
     status: (IN/OUT)  MDAS_status
```

(This needs to be updated for graphs.) `MDAS_GRAPH_CURSOR_SCAN()` returns the `cursor` position, `level`, or `size` values of `MDAS_graph` structures. The result is returned in `scalar`. The `gh` argument is a handle to the structure of interest. Valid `action` values are:

MDAS_LEVEL_GET get the level of the current cursor position
MDAS_CURSOR_SIZE get the size of the MDAS_graph structure at the current level
MDAS_CURSOR_GET get index of current cursor position in current level

```
MDAS_GRAPH_CURSOR(gh, action, index, status)
     gh:     (IN/OUT)  MDAS_GRAPH_HANDLE
     action: (IN)      integer
     index:  (IN)      integer
     status: (IN/OUT)  MDAS_status

MDAS_GRAPH_COPY(gh, ghcopy, status)
     gh:     (IN/OUT)  MDAS_GRAPH_HANDLE
     ghcopy: (OUT)     MDAS_GRAPH_HANDLE
     status: (IN/OUT)  MDAS_status

MDAS_GRAPH_EXEC(gh, rsrc, status)
     gh:     (IN/OUT)  MDAS_GRAPH_HANDLE
     rsrc:   (IN)      MDAS_INFOH
     status: (IN/OUT)  MDAS_status

MDAS_GRAPH_FREE(gh, status)
     gh:     (IN/OUT)  MDAS_GRAPH_HANDLE
     status: (IN/OUT)  MDAS_status
```

4.8.2.4 Direct Query Interface

```
MDAS_SD_QUERY(info, cond, extent, result, status)
     info:   (IN)      MDAS_INFOH
     cond:   (IN)      MDAS_INFOH
     extent: (IN)      MDAS_INFOH
     result: (OUT)     MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

MDAS_SD_QUERY() extends the functionality of MDAS_INQUIRE() to user-defined *specific data*. The call may fail if no methods exist to perform the requested query on the specified data set. Where supported, MDAS will search the contents of user data for occurences of user tokens and values specified in info. An error is returned if no such tokens are found.

```
MDAS_SQL_EXEC(...)
     ... TBD
```

Since MDAS targets interoperation between many protocols, an ANSI SQL execute call is also provided for use with SQL-enabled servers opened with MDAS_CONNECT().

4.8.2.5 Table Interface

The MDAS Library provides and interface for the direct import and export of DBMS tables.

TBD.

Input/Output operations on large tables may be internally spooled.

4.9 MDAS Metadata

This section provides details on the type of metadata maintained by MDAS. It also describes a few scenarios that show the use of this metadata. MDAS maintains metadata to support the following functionality:

1. locating entities
2. gaining access to entities, and
3. performing computations with the entities

4.9.1 Locating Entities

The metadata associated with the *locator* functionality provides for a variety of ways for locating entities in the information space. In the case of data set elements, locator metadata may originate either from portions of the element itself (implicit metadata) and/or may arise from other parametric values (explicit metadata) that aid in locating the element. For example, the contents of a document form part of implicit metadata whereas its name and the semantics of the contents form part of explicit metadata. In the case of methods, resources and users, the associated metadata is typically explicit in nature. Entities that are of interest are identified by queries against either the system metadata (attributes stored for all entities) or entity metadata (attributes that are specific to that class of entity).

4.9.2 Accessing Entities

The *access* metadata provides information on how to transport an entity, e.g. data set, from its current location to a location of interest. Data sets and methods are example of transportable entities. In the case of non-transportable entities, the *access* metadata provides information on how to communicate with the entity. Resources and users are examples of such entities. For transportable entities, the metadata provides information on how to authenticate and connect to the corresponding data server, and the method to use to perform and mediate the transfer operation. Information on the applicable levels of locking are maintained to control the amount of concurrent activity. Metadata about constraints that are imposed on the transfer operation can also be maintained. Transportable entities may exist in replicates, or may be distributed over a network, or possibly both. Transport methods may allow for parallel transport. Metadata to determine which replicate can be cost-effectively transferred and metadata to formulate a plan to perform parallel access of data sets may also be stored in required cases. In the case of non-transportable entities, the metadata provides methods which one can use to communicate with the entity and also the parametric values that can be used to establish this communication. Again, authentication and security metadata may also be maintained in certain cases. In the case of resources, statistical and usage metadata is maintained to facilitate scheduling and accounting applications.

4.9.3 Computing With Entities

For data sets, the *compute-oriented* metadata provides information on how to transform and present the data to various processes, users and resources. A data set may be stored in one form while a computing process may require the data in another form. MDAS can provide the transformation to the acceptable form by maintaining metadata about this procedure. In the case of resources and methods, the compute-oriented metadata provides information about the characteristics of inputs and outputs (e.g. ability to support parallel I/O), and auxiliary support structures needed for effective computation.

4.9.4 Usage of Metadata

Following are examples of the types of requests that MDAS can respond to based on its metadata.

4.9.4.1 Locating Data Sets

A user is only able to describe a data set of interest, but does not know its location. For example:

- Locate the data set that was created on 10/10/96 by using the Spectral Gradient Transform on the Intel Paragon with wave data collected by EOS satellite at 2200 hrs.
- Locate the text document created by the OCR program based on manuscript page 24 for the patent on "Multithreaded Supercomputers".
- Locate the document provided by Intel about the Paragon and dealing with FIFO cache registers.

4.9.4.2 Locating Resources

A user needs resources which conform to some user-defined specification but is not aware of the actual set of resources available in the heterogeneous, distributed computing environment. For example:

- Find a computing platform with a 100 Teraflops rating which is connected to Suranet
- Find a computing platform which can run the "Smith-Waterson" algorithm and also provide 1 Teraflops for a 20-minute interval in the next 24-hour period.
- Find the nearest available resource that holds text and image files for Patent number "22224444". Execute the patent conversion utility that translates the files into LaTeX and *tiff* form.

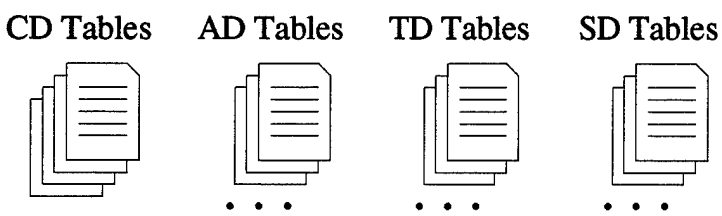

Figure 4.3: MDAS Metadata Tables

4.9.4.3 Locating Methods

A user needs to find out if a specific computational function (method) exists or can be applied to some data set(s) using specified resource(s). For example:

- Find a format conversion method to convert from the netCDF format to the vis5D format.
- Is there a method to validate 3D ECOM-si hydrodynamic ocean simulation output with observational data for the San Diego Bay?
- Can I run a parallel interpolation function on regular grids of size greater than 1024 x 1024 on a CRAY MPP platform?

4.9.5 Metadata Schema

The metadata described in the previous sections is stored in the MDAS Metadata Catalog shown in Figure 4.3. The central component of the metadata database schema consists of four main *Catalog Data (CD)* metadata tables, one each for the main elements of MDAS, viz. datasets, methods, resources and users. The CD metadata tables are supported by several companion *Auxiliary Data (AD)* metadata tables. The CD and AD metadata tables form a normalized schema for the metadata database. The definitions and descriptions of several of the attribute values in CD and AD metadata tables are contained in *Token Data (TD)* metadata tables. Apart from these three types of metadata tables, which form the system level component of the metadata database, optional *Specific Data (SD)* metadata tables can be defined for each user, dataset, method, and resource in the system. The SD metadata tables make up the application level component of the metadata database. Provisions are made to access these tables in conjunction with the system level metadata. All or parts of these tables may be replicated in more than one metadatabases for high availability and performance. The compositions of and interactions between the table are described below.

Table 4.7 provides the types of attributes used in the metadata schema. Table 4.8 provides the attributes of all TD metadata tables. Table 4.9 provides the attributes of all CD metadata tables. Tables 4.10, 4.11, 4.12 and 4.13 provide AD metadata for datasets, methods,

resources and users in MDAS. The schema digram for part of the metadata database is given in Figure 4.4.

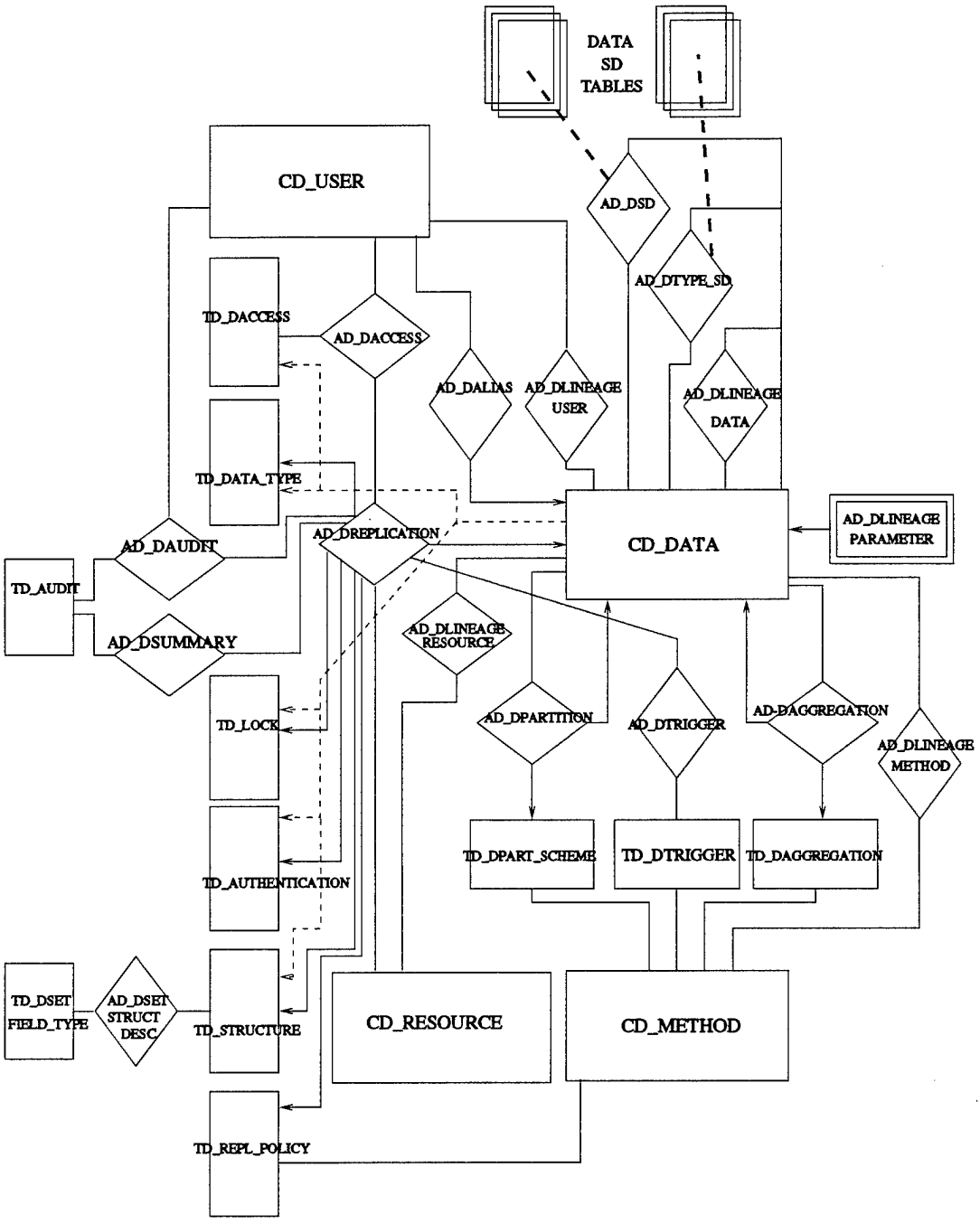

Figure 4.4: Metadata Data Model (data-centric)

native type
MDAS_DATA_ID
MDAS_RSRC_ID
MDAS_METH_ID
MDAS_USER_ID
MDAS_DATA_TYPE_ID
MDAS_RSRC_TYPE_ID
MDAS_METH_TYPE_ID
MDAS_USER_TYPE_ID
MDAS_DATA_NAME
MDAS_RSRC_NAME
MDAS_METH_NAME
MDAS_USER_NAME
MDAS_DATA_TYPE_NAME
MDAS_RSRC_TYPE_NAME
MDAS_METH_TYPE_NAME
MDAS_USER_TYPE_NAME
MDAS_USER_ADDRESS
MDAS_USER_PHONE
MDAS_USER_EMAIL
MDAS_USER_DOMAIN

native type
MDAS_FIELD_NAME
MDAS_FIELD_TYPE
MDAS_FIELD_TYPE_NAME
MDAS_PATH_NAME
MDAS_REP_POLICY_DESC
MDAS_REP_TYPE_DESC
MDAS_PARTITION_DESC
MDAS_TRIG_DESC
MDAS_TRIG_MODE
MDAS_TRIG_CONDITION
MDAS_ACCESS_DESC
MDAS_PRED_DESC
MDAS_FUNCTION_DESC
MDAS_FUNCTION_NAME
MDAS_LANG_TYPE
MDAS_SCHEMA_ID
MDAS_ACCESS_ID
MDAS_TRIGGER_ID
MDAS_AGGREGATION_ID
MDAS_FUNCTION_ID
MDAS_LOCATION_ID
MDAS_LOC_DESC

native type
MDAS_TICKET_DESC
MDAS_AUTHENTICATION_DESC
MDAS_LOCK_DESC
MDAS_MISC_TYPE_ID
MDAS_LOCK_ID
MDAS_TICKET_ID
MDAS_AUTHENTICATION_ID
MDAS_MISC_VALUE
MDAS_MISC_NAME
MDAS_REP_POLICY_ID
MDAS_REP_TYPE_ID
MDAS_PARTITION_SCHEME_ID
MDAS_ACTION_ID
MDAS_ACTION_DESC
MDAS_PARAMETER_TYPE
MDAS_PARAMETER_VALUE
MDAS_PARAMETER_NAME
MDAS_SD_DESC
MDAS_KEY
MDAS_SCHEMA_TYPE_NAME
MDAS_METH_EXEC_TYPE_NAME
MDAS_METH_EXEC_TYPE_ID

Table 4.7: Native Type Definitions (MDAS Mid-level)

table name	*attributes*
MDAS_TD_LOCK	lock_id, lock_description, lock_method_id
MDAS_TD_DOMAIN	domain_id, domain_description
MDAS_TD_DATA_TYPE	mo_data_type_id, data_type_name, parent_data_type_id
MDAS_TD_METH_TYPE	mo_method_type_id, method_type_name, parent_method_type_id,
MDAS_TD_RSRC_TYPE	mo_resource_type_id, resource_type_name, parent_resource_type_id
MDAS_TD_USER_TYPE	mo_user_type_id, user_type_name, parent_user_type_id
MDAS_TD_EXEC_TYPE	executable_type_id, executable_type_name
MDAS_TD_LOCATION	location_id, location_description, country, state, city, county, site, building, wing, floor, room, organization, department, division, subdivision, project, domain, group, latitude, longitude, timezone, zipcode, netprefix, nettype,
MDAS_TD_RSRC_FUNCTION	function_id, function_description, function_compliance_description, function_name
MDAS_TD_DSET_FIELD_TYPE	field_type, field_type_name, conform_language
MDAS_TD_DSET_SCHEMA	mo_schema_id, schema_name, parent_schema
MDAS_TD_DSET_SCHEMA_DESCRIPTION	mo_schema_id , field_enum , field_name , field_type
MDAS_TD_SPECTRAL_SUMMARY	spectral_id, spectral_description
MDAS_TD_ACTION	action_id, action_description
MDAS_TD_TICKET	ticket_id, ticket_description, ticket_method
MDAS_TD_REPLICATION_TYPE	replication_type_id, description
MDAS_TD_AUTHENTICATION	authentication_id, authentication_description, authentication_name, authentication_kind, authentication_keylength, authentication_public_info, authentication_metod_id
MDAS_TD_VERIFICATION	verification_id, verification_description, verification_name, verification_kind, verification_keylength, verification_public_info, verification_metod_id
MDAS_TD_ENCRYPTION	encryption_id, encryption_description, encryption_name, encryption_kind, encryption_keylength, encryption_public_info, encryption_metod_id
MDAS_TD_DECRYPTION	decryption_id, decryption_description, decryption_name, decryption_kind, decryption_keylength, decryption_public_info, decryption_metod_id
MDAS_TD_RSRC_ACCESS	mo_access_id, access_constraint, ticket_id
MDAS_TD_METH_ACCESS	mo_access_id, access_constraint, access_ticket_id
MDAS_TD_DSET_ACCESS	mo_access_id, access_constraint, access_ticket_id
MDAS_TD_DSET_REPLICATION_POLICY	replication_policy_id, replication_description, replication_policy_method_id
MDAS_TD_DSET_PARTITION_POLICY	partition_scheme_id, partition_description, partition_policy_method_id
MDAS_TD_DSET_TRIGGER	mo_trigger_id, trigger_description, trigger_method_id, trigger_mode, trigger_condition, destination_data_id
MDAS_TD_DSET_AGGREGATION	mo_aggregation, aggregation_description, aggregation_method
MDAS_TD_METH_REPLICATION_POLICY	replication_policy_id, replication_policy_description, replication_policy_method_id
MDAS_TD_RSRC_REPLICATION_POLICY	replication_policy_id, replication_policy_description_id, replication_policy_method_id

Table 4.8: Type Definition Tables (MDAS Mid-level)

table name	*attributes*
MDAS_CD_DATA	mo_data_id, mo_data_name, global_data_type_id, global_schema_id, verification_key , verification_id, is_partitioned
MDAS_CD_METHOD	mo_method_id, mo_method_name, mo_method_type_id, verification_key, verification_id, public_key, public_id, is_compound
MDAS_CD_RESOURCE	mo_resource_id, mo_resource_name, mo_resource_type_id, verification_key, verification_id, public_key, public_id
MDAS_CD_USER	mo_user_id, mo_user_name, mo_user_address, mo_user_phone, mo_user_phone2, mo_user_fax, mo_user_email, mo_user_domain, mo_user_type_id, verification_key, verification_id, public_key, public_id

Table 4.9: Catalog Data Tables (MDAS Mid-level)

table name	*attributes*
MDAS_AD_DSET_ALIAS	mo_data_id, mo_data_name, mo_user_id
MDAS_AD_DSET_DOMAIN	mo_data_id, domain_id
MDAS_AD_DSET_AUTHENTICATION_KEY	mo_data_id, private_key, private_lockbox_method, authentication_id
MDAS_AD_DSET_REPLICATION	mo_data_id, replication_enum, mo_data_name, mo_data_type_id, mo_schema_id, path_name, mo_resource_id, replication_type_id, replication_policy_id, replication_timestamp, size, cardinality, is_deleted, permanence, default_flags
MDAS_AD_DSET_LOCK	mo_data_id, replication_enum, mo_user_id, start_time, end_time, lock_id
MDAS_AD_DSET_ACCESS	mo_data_id, replication_enum, mo_user_id, mo_access_id
MDAS_AD_DSET_PARTITION	mo_data_id, partition_data_id, partition_enum, partition_scheme
MDAS_AD_DSET_AGGREGATION	mo_data_id, aggregation_id, aggregation_data_id
MDAS_AD_DSET_SUMMARY	mo_data_id, replication_enum, action_id, spectral_timestamp, spectral_id, spectral_values
MDAS_AD_DSET_TRIGGER	mo_data_id, replication_enum, validated_condition, mo_trigger_id
MDAS_AD_DSET_AUDIT	mo_data_id, replication_enum, mo_user_id, action_id, time_stamp
MDAS_AD_DSET_LINEAGE_DATA	mo_data_id, replication_enum, parent_data_id, parent_in_data_enum
MDAS_AD_DSET_LINEAGE_METHOD	mo_data_id, replication_enum, parent_method_id, child_output_enum
MDAS_AD_DSET_LINEAGE_USER	mo_data_id, replication_enum, owner_user_id
MDAS_AD_DSET_LINEAGE_PARAMETER	mo_data_id, replication_enum, parameter_enum, parameter_value
MDAS_AD_DSET_LINEAGE_RESOURCE	mo_data_id, replication_enum, sub_method_enum, parent_resource_id
MDAS_AD_DSET_SD	mo_data_id, sd_description, sd_data_id
MDAS_AD_DSET_TYPE_SD	mo_data_type_id, sd_description, sd_data_id

Table 4.10: Auxiliary Data Tables for Datasets (MDAS Mid-level)

table name	*attributes*
MDAS_AD_METH_ALIAS	mo_method_id, mo_method_name, mo_user_id
MDAS_AD_METH_DOMAIN	mo_method_id, domain_id
MDAS_AD_METH_AUTHENTICATION_KEY	mo_method_id, private_key, private_lockbox_method, authentication_id
MDAS_AD_METH_DECRYPTION_KEY	mo_method_id, private_key, private_lockbox_method, decryption_id
MDAS_AD_METH_REPLICATION	mo_method_id, replication_enum, mo_method_name, mo_method_type_id, executable_type_id, mo_resource_id, path_name, replication_type_id, replication_policy_id, replication_timestamp
MDAS_AD_METH_LOCK	mo_method_id, replication_enum, mo_user_id, start_time, end_time, lock_id
MDAS_AD_METH_ACCESS	mo_method_id, replication_enum, mo_user_id, mo_access_id
MDAS_AD_METH_SUMMARY	mo_method_id, replication_enum, action_id, spectral_timestamp, spectral_id, spectral_value
MDAS_AD_METH_AUDIT	replication_enum, mo_method_id, mo_user_id, action_id, time_stamp
MDAS_AD_METH_LINEAGE_METHOD	mo_method_id, replication_enum, parent_method_id, child_output_enum
MDAS_AD_METH_LINEAGE_DATA	mo_method_id, replication_enum, parent_data_id, parent_in_data_enum
MDAS_AD_METH_LINEAGE_USER	mo_method_id, replication_enum, owner_user_id
MDAS_AD_METH_LINEAGE_PARAMETER	mo_method_id, replication_enum, parameter_enum, parameter_value
MDAS_AD_METH_LINEAGE_RESOURCE	mo_method_id, replication_enum, sub_method_enum, parent_resource_id
MDAS_AD_CONVERT_METHOD_ID	mo_method_id, in_data_type_id, out_data_type_id
MDAS_AD_CONVERT_METHOD_TYPE	mo_method_type_id, in_data_type_id, out_data_type_id
MDAS_AD_METH_APPLICATION_PARAMETER	mo_method_id, parameter_enum, parameter_type, parameter_name, parameter_default_value
MDAS_AD_METH_APPLICATION_OUTPUT	mo_method_id, out_enum, out_data_type, out_data_name, out_default_value
MDAS_AD_METH_APPLICATION_INPUT	mo_method_id, in_enum, in_data_type, in_data_name, in_default_value
MDAS_AD_METH_APPLICATION_REQUIREMENTS	mo_method_id, misc_name, misc_type, misc_value
MDAS_AD_METH_APPLICATION_PREDICTION	mo_method_id, replication_enum , prediction_method_id, prediction_description
MDAS_AD_METH_COMPOUND_METHOD_MAP	mo_method_id, method_enum, sub_method_id
MDAS_AD_METH_COMPOUND_DSET_MAP	mo_method_id, producer_method_enum, producer_out_enum, consumer_method_enum, consumer_in_enum
MDAS_AD_METH_COMPOUND_PARAMETER_MAP	mo_method_id, method_param_enum, sub_method_param_enum
MDAS_AD_METH_SD	mo_method_id, sd_description, sd_data_id
MDAS_AD_METH_TYPE_SD	mo_method_type_id, sd_description, sd_data_id

Table 4.11: Auxiliary Data Tables for Methods (MDAS Mid-level)

table name	*attributes*
MDAS_AD_RSRC_ALIAS	mo_resource_id, mo_resource_name, mo_user_id
MDAS_AD_RSRC_DOMAIN	mo_resource_id, domain_id
MDAS_AD_RSRC_AUTHENTICATION_KEY	mo_resource_id, private_key, private_lockbox_method, authentication_id
MDAS_AD_RSRC_DECRYPTION_KEY	mo_resource_id, private_key, private_lockbox_method, decryption_id
MDAS_AD_RSRC_REPLICATION	mo_resource_id, replication_enum, mo_resource_name, mo_resource_type_id, location_id, replication_type_id, replication_timestamp
MDAS_AD_RSRC_LOCK	mo_resource_id, replication_enum, mo_user_id, start_time, end_time, lock_id
MDAS_AD_RSRC_ACCESS	mo_resource_id, replication_enum, mo_user_id, mo_access_id
MDAS_AD_RSRC_SUMMARY	mo_resource_id, action_id, replication_enum, spectral_timestamp, spectral_id, spectral_value
MDAS_AD_RSRC_AUDIT	mo_resource_id, replication_enum, mo_user_id, action_id, time_stamp
MDAS_AD_RSRC_LINEAGE_RESOURCE	mo_resource_id, replication_enum, sub_method_enum, parent_resource_id
MDAS_AD_RSRC_LINEAGE_METHOD	mo_resource_id, replication_enum, parent_method_id
MDAS_AD_RSRC_LINEAGE_USER	mo_resource_id, replication_enum, owner_user_id
MDAS_AD_RSRC_LINEAGE_PARAMETER	mo_resource_id, replication_enum, parameter_enum, parameter_value
MDAS_AD_RSRC_LINEAGE_DATA	mo_resource_id, replication_enum, parent_data_id, parent_in_data_enum
MDAS_AD_FUNCTION_ON_RESOURCE	mo_resource_id, mo_method_id, function_id
MDAS_AD_RSRC_APPLICATION_PREDICTION	mo_resource_id, replication_enum, prediction_method_id, prediction_description
MDAS_AD_RSRC_SD	mo_resource_id, sd_description, sd_data_id
MDAS_AD_RSRC_TYPE_SD	mo_resource_type_id, sd_description, sd_data_id

Table 4.12: Auxiliary Data Tables for Resources (MDAS Mid-level)

table name	*attributes*
MDAS_AD_USER_ALIAS	mo_user_id, mo_user_name
MDAS_AD_USER_GROUP	mo_user_id, group_user_id
MDAS_AD_USER_DOMAIN	mo_user_id, domain_id
MDAS_AD_USER_AUTHENTICATION_KEY	mo_user_id, private_key, private_lockbox_method, authentication_id
MDAS_AD_USER_DECRYPTION_KEY	mo_user_id, private_key, private_lockbox_method, decryption_id
MDAS_AD_USER_SUMMARY	mo_user_id, action_id, spectral_timestamp, spectral_id, spectral_value
MDAS_AD_USER_AUDIT	mo_user_id, action_id, time_stamp
MDAS_AD_USER_SD	mo_user_id, sd_description, sd_data_id
MDAS_AD_USER_TYPE_SD	mo_user_type_id, sd_description, sd_data_id

Table 4.13: Auxiliary Data Tables for Users (MDAS Mid-level)

We start with the CD table for datasets along with two important AD tables.

```
MDAS_CD_DATA
    (mo_data_id                 %primary key
    mo_data_name
    global_data_type_id         %references MDAS_TD_DATA_TYPE
    global_schema_id            %references MDAS_TD_DSET_STRUCTURE
    verification_key            % public key for verification
    verification_id             %references MDAS_TD_VERIFICATION
    is_partitioned
    );
MDAS_AD_DSET_REPLICATION
    (mo_data_id                 %references MDAS_CD_DATA
    replication_enum            %unique to each copy of a dataset
    mo_data_name
    mo_data_type_id             %references MDAS_TD_DATA_TYPE
    mo_schema_id                %references MDAS_TD_DSET_STRUCTURE
    path_name                   %path of file/large object or database name of table
    mo_resource_id              %references MDAS_CD_RESOURCE
    replication_type_id         %references MDAS_TD_DSET_REPLICATION_TYPE
    replication_policy_id       %references MDAS_TD_DSET_REPLICATION_POLICY
    replication_timestamp
    size
    cardinality
    is_deleted
    permanence
    default_flags
    );
MDAS_AD_DSET_PARTITION
    mo_data_id                  %references MDAS_CD_DATA
    partition_data_id           %references MDAS_CD_DATA
                                %unique ids given to each partition
    partition_enum              %enumeration of partition of a dataset
    partition_scheme_id         %references MDAS_TD_DSET_PARTITION_POLICY
    );
```

The *mo_data_id* is a catalog-wide unique identifier given to each dataset registered in the catalog. We allow two separate MDAS catalog systems to share the same identifier space; but, we ensure uniqueness across all catalogs by combining data set identifiers with the catalog names. Hence, we do not require a centralized identifier-generator system to maintain unique identity. The catalog name for each MDAS catalog is generated using the ip or network address of the system on which the MDAS resides and also the creation timestamp, thereby ensuring uniqueness in the MDAS catalog names domain. In MDAS, the identifiers for datasets are generated using an incremental counter (stored in a table of counters). One can ask for the next identifier or ask for a block of n identifiers.

Even though the combination of dataset identifiers and catalog names provide uniqueness in identification, it does not solve the problem of finding whether two identifiers (residing in two

different catalogs) point to the same dataset. That resolution can be done by dereferencing the *path_name* and resolving the *mo_resource_ids*. We consider that this problem of dual references is not important enough to warrant a complicated method to generate unique identifiers based on contents and/or location.

Normally, deleted datasets do not relinquish their identifiers and the identifiers are not reused. But, provisions will be made to provide a high water mark (possibly around the 90% mark of the maximum possible identifier that can be generated) in the identifier generation when a garbage collection is performed to get a list of reusable identifiers. In case, one hits a second higher (98%) water mark (possibly after reusing old identifiers), MDAS should start building another catalog and redirect all insertions to the new catalog.

A registered dataset can either be replicated or partitioned. Each replicated copy retains the *mo_data_id* given to the dataset in the MDAS_CD_DATA, whereas each partition of a dataset are given a unique *mo_data_id*. This decision was made because of the following reasons: each partition of a database can be manipulated separately independent of other partitions and hence behave as individual datasets in their own rights. On the other hand, even though each replication of a dataset are accessed independently of other replicas, any changes made to one should be reflected in all of them, i.e., each replica should be semantically isomorphic; hence they carry the same identifier.

Each replica is identified from its siblings by the *replication_enum* attribute as given in table MDAS_AD_DSET_REPLICATION. The attribute *mo_data_id* is a virtual identifier, where as the combination (*mo_data_id, replication_enum*) forms a real identifier. The (*mo_resource_id*, *path_name*) combination provides the physical means of locating the actual dataset. Even though each replica are required to be semantically equivalent, they can differ in their formats (syntax). For example, consider a latex file **paper.tex** and its derivatives **paper.dvi** and **paper.ps**. In MDAS, the three files are considered to be semantically equivalent and are stored under the same identifier. The *mo_data_name* and *mo_data_type_id* attributes defined in the MDAS_AD_DSET_REPLICATION allows for different type and name values. We do not allow the concept of partitioning a replica.

The *is_partitioned* attribute in the MDAS_CD_DATA table flags whetehr the dataset is partitioned or not. Unique identification of partitions is done in the MDAS_AD_DSET_PARTITION. The *partition_enum* attribute provides the actual ordering of the partitions with respect to the original unpartitioned dataset. Partitions provide convenience in more than one way. For very large datasets, finding space in a single storage system may not be possible and partitioning provides a natural means to overcome this problem. Partitions can be driven by application-level constraints and for efficiency reasons. Assume a dataset that contains nation-wide data, where the data is mostly used inside regions with a few operations performed nation-wide. Then partitioning the data region-wise, and distributing them to be stored at regional centers may be highly efficient. Since the datasets have distinct identifiers, updates, access, locks and other access controls on one of the partitions need not affect other partitions. An operation over the full dataset can still be accomplished seamlessly by performing it on the (virtual) parent dataset which will be automatically applied to all the partitions.

Partitioning the dataset is controlled by a *partition_policy_method_id* attribute; the table MDAS_TD_DSET_PARTITION_POLICY provides a pointer to this method. This affords

a flexibility to perform the partitioning using syntactic or semantic criteria and is not dependent on partitioning provided by the storage system. For example, a large text, such as a patent, can be partitioned such that all its images are stored in one partition and other parts in another partition. One may then view or modify only the text parts, or the image parts, without accessing the other parts.

Each partition of a dataset can be of different types since they are viewed as registered datasets. This mechanism allows one to store each partition in a convenient form or in an efficient form at the storage site. For example, a large table that is partitioned and distributed nation-wide may be stored in a variety of database systems taking advantage of what is available locally; one partition may be stored in a DB2 system, where as another may reside in an Oracle database, and yet another in an Illustra database. Such storage flexibility may allow local user to access the data in a method familiar and convenient to them. Operations on other partitions or on all partitions may seamlessly present the table in a familiar form by making appropriate changes based on the *global_data_type_id* attribute of each partitioned dataset. Note that, in the case of database tables, this functionality is an enhancement on the ODBC concept as it applies to partitions of datasets which are of different types instead of datasets of different types. The corresponding operation (of seamless integration of objects of different types) for datasets of other types (other than database tables) are unique to MDAS.

If an inverse operation for a partition policy method exists (this information is registered in MDAS_AD_CONVERT_METHOD_ID table) then one can use it as a convenient way to recreate back a singe dataset. The *partition_policy_method_id* also plays another important role. In the case of reads or update access, the policy method provides a way to identify the partition (or partitions) on which the operation need to be performed. Hence, the policy method is used not only as a means of partitioning but also as a access redirection mechanism. In the current design of the MDA system, one can partition each dataset using only one method. That is, more than one way of partitioning a dataset is not allowed. But each partition can be further partitioned if necessary and further, any partition can also be replicated if desired.

The *replication_policy_id* attribute in MDAS_AD_DSET_REPLICATION table provides the method (a pointer to which is stored in the MDAS_TD_DSET_REPLICATION_POLICY table) that is used for updating purposes. That is, whenever a replica is changed, this method should be invoked to ensure that the updates are propagated if desired. The updating method can be different for each replica and provides a rich semantics of replication. One can perform an immediate replication or deferred replication depending upon which copy is being updated. Moreover, one can control which types of updates are passed to other replicas immediately, and which types of updates are batched to be deferred for later transmission. Updates can even be controlled from being invisible to other replicas by having a null update method thus providing a version capability. One may also have updates to be propagated only on explicit commands by the user or only at system-defined intervals or events. The update methods can perform the necessary checks before passing on the updates to its replicas. The user is also notified of any errors raised during the updates.

Another parameter that comes into the picture during updates on the replicas is the *replication_type_id* attribute of the replicas. We envisage that replication may be done

in many ways. One can perform a *master-slave* copy which allows one to update only the master copy which is then passed on to the slave copies; one can perform a *peer-to-peer* copy where updates can occur on any of the copies and can be passed onto others; one can perform a *private copy* where changes are allowed on the private copy but are not passed along to other copies and updates at other places will not be reflected in the private copy; one can have a *backup* copy where no updates are allowed and the copy is also not visible to the users unless explicitly requested; one can have a *version* copy where updates are allowed on the new version and the old version will not be updated though it is available to other user groups; one can have *divergent* copies where no action is taken on updates and each copy diverges from its peers, and a synchronizing update is performed at a later time to reconcile the differences, or one can have a *transient* copy which can vanish after some time and is not shown to users who query about the copies. Other esoteric updates can be similarly accommodated using the combination of *replication_policy_id* and *replication_type_id* attributes.

The *mo_data_types* of datasets form an hierarchy (see MDAS_TD_DATA_TYPE table). Actually types of all entities registered by MDAS have this property. This allows for inheritance of properties and functions in the hierarchy. We do not provide explicit functionalities and properties for datasets (or other entities) as part of the system-level metadata; we consider them to be part of specific data and expect SD tables to be defined for these purposes.

The *mo_data_type_id* attribute in the MDAS_AD_DSET_REPLICATION table also provides an ability to replicate a database in different formats. Updates on such replicates in the MDA system generalizes the concept of traditional updates in distributed systems. Apart from update operations such as insertions, deletions and modifications on texts, tables or other objects being mirrored in other copies of the same type, here one also needs to contend with updates across datasets of different types. An update performed on a dataset of particular type may require regeneration of a new copy of the replica using thealternate format.

Revisiting the example of the latex file **paper.tex** and its derivatives **paper.dvi** and **paper.ps**, any changes to **paper.tex** can be reflected on its **dvi** replica by running a *latex* command (or a make command if other files are also involved) on the updated **paper.tex** file. Further the **paper.ps** file may be regenerated from the **paper.dvi** file through a *dvi2ps* command. These two commands may be packaged as a single update method for the two files. In this case a useful method of replication is that the latex file is stored as a *master* and the other two files are stored as its slave replicas. The mirrored updates to the derived files may be done at a user given explicit update command, or at significant intervals.

Note that with this semantics, the update command is being used to provide a form of *method transparency*, wherein the commands to perform *latex* and the *dvi2ps* tasks are hidden from the user as an update method. Similar transparencies can be accomplished for compilation and linking, for generating up-to-date statistics, for real-time aggregation or deriving operations, for real-time dataset displays under update and even for printing a dataset (where a particular replica is considered to be stored in the printer in a printable format!!).

One can provide real-time visualization by registering a *transient* replica of the dataset as being "stored" on the screen through an *interactive viewer*. The update method issues a

redraw command to the viewer. Note that in this case, the screen is viewed as a resource that can be used as a write-only transient store. The *is_deleted* attribute can be used to control the transience of the replica. For example, consider that the user is viewing the **paper.tex** file using an *interactive viewer* on its **paper.ps** replica, and has registered this "view" as a *transient* replica. Then, any updates to **paper.tex** gets reflected on its **paper.dvi** and **paper.ps** replica file and further on the screen if proper reopen commands are placed in its update method. Similarly, in this seamless semantics, the notion of a 'tele-conferencing' or 'designing-by-group', and 'tele-operating' reduce to that of update performance.

The *replication_policy_id* and *replication_type_id* attributes also provide information about the consistency and currency of a replica with respect to its peers. Hence, one can access a copy knowing whether it contains stale data. One is given a choice of accessing datasets based on allowed levels of staleness (ascertained by knowing the replication policy of the replicas) and use this criteria to access a stale replica that may be cheaper to access (possibly it resides on a local file store) than to access another replica that is current but needs a costly access operation.

The replication paradigm defined in MDAS also helps in providing a a different form of *format transparency.* Again using the 'paper' example, one can issue an *edit* command on the **paper** dataset and the system will use the **paper.tex** file as the target file. If a print command is used on the same **paper** dataset, the **paper.ps** file will be used to spool to a (Postscript) printer.

Next we discuss the role of the *schema* attribute in MDAS_AD_DSET_REPLICATION and MDAS_CD_DATA tables. Tables MDAS_AD_DSET_STRUCTURE_DESCRIPTION and MDAS_TD_DSET_STRUCTURE describes the schema of the dataset. For example, in a database table it will provide the database schema for the table and for a text file, it may provide the sectional organizations. One can use the schema description to select or read portions of interest from the dataset control mechanisms.

We allow the schema attribute at the MDAS_CD_DATA level in order to provide a place to hold a schema independent of the individual copy format. For example, for a table replicated in a variety of database systems, the *global_schema_id* can be used to provide a generic schema. We also allow hierarchies in the schema structure. Not only can this be used as an inheritance mechanism for properties of schema attributes, but it can also be used to record schema evolution.

We discuss the *verification* attributes when we discuss MDAS_CD_METHOD.

In MDAS REPLICATION AD tables are also defined for methods and resources. Since a method can be replicated (possibly compiled for different platforms) such a table is necessary. The reason for having a REPLICATION AD table for resources is to provide a kind of *resource transparency.* In an organization, one may view all printers as common resource and a print command can choose any one of the available ones; these printers are viewed to be functionally (semantically) equivalent. Of course, the notion of updates does not exist for resources and the replication policy is used to note how to replicate a given resource (eg. make and configuration files needed for setting up an identical resource).

The PARTITION AD table is particular to datasets and has no equivalent counterpart for

methods, resources and users.

The following AD tables along with relevant TD tables capture access control metadata for datasets in MDAS.

MDAS_AD_DSET_ACCESS
(mo_data_id *%together form reference to*
replication_enum *%MDAS_AD_DSET_REPLICATION*
mo_user_id *%references MDAS_CD_USER*
mo_access_id *%references MDAS_TD_DSET_ACCESS*
);
MDAS_AD_DSET_LOCK
(mo_data_id *%together form reference to*
replication_enum *%MDAS_AD_DSET_REPLICATION*
mo_user_id *%references MDAS_CD_USER*
start_time
end_time
lock_id *%references MDAS_TD_LOCK*
);
MDAS_AD_DSET_DOMAIN
(mo_data_id *%references MDAS_CD_DATA*
domain_id *%references MDAS_TD_DOMAIN*
);

The *mo_access_id* attribute can be used to find the types of access (as described in the MDAS_TD_DSET_ACCESS table) that are allowed on the dataset and can also be used to obtain an appropriate ticket once the appropriate access has been validated. The MDAS_AD_DSET_ACCESS table uses the user's identification to check for access permissions to a dataset. The types of access that are permitted are defined by the *access_constraint* attribute in the MDAS_TD_DSET_ACCESS table. Examples of access include read, write, append, delete, execute, replicate, partition, control, etc. If a type of access to a dataset for a particular user is permitted a corresponding entry will be available in the AD table Then, the *ticket_id* attribute (in the TD table) can be used to obtain a ticket that will provide a *user transparent* access to the dataset in a particular domain. Such a ticket can be used to validate the access to a dataset if a remote storage system that stores the dataset requires user authentication.

The *lock* attribute is primarily for transaction management purposes. Its utility is similar to that found in database management systems. Apart from the normal kinds of locks (eg. read lock, exclusive read lock, write lock, etc.,), the *lock* attribute can also be used as a reservation mechanism. Assume that one of the users is planning to edit a copy of the *paper.tex* file and wishes that it not be used by others during that period. The user can lock the replica of interest for sufficient amount of time. This facility will be useful when one needs to lock all copies of a dataset for some extended time for some purpose. Then, other users are given an indication about when to expect the dataset to be back on line. The lock facility can also be used for advance reservation. One can anticipate one's needs and reserve the dataset for a block of time in the future. At the dataset level this functionality will provide a means to facilitate *group_design* with individual time slots. In

the case of resources and methods, the reservation via locking mechanism gives a facility to ensure future availability of resources for quality of service purposes, and can be used by schedulers effectively.

The concept of a *domain* is used as an additional level of security, apart from the ones given by access, and authentication and encryption mechanisms. A domain is considered to be a secure area in which one can expect protection with respect to authenticated datasets and secure communications. We consider that each dataset is associated with one or more domains (whose descriptions are given in the MDAS_TD_DOMAIN table) authorized in the MDAS_AD_DSET_DOMAIN table. One, can move a dataset, or parts or copies of it, only within those domains.

In MDAS, ACCESS and LOCK AD tables are also provided for methods and resources, and DOMAIN AD table is provided for users, methods and resources.

Next we look at two AD tables useful for statistical and accounting purposes.

```
    );
MDAS_AD_DSET_AUDIT
    (mo_data_id             %together form reference to
    replication_enum        %MDAS_AD_DSET_REPLICATION
    mo_user_id              %references MDAS_CD_USER
    mo_action_id            %references MDAS_TD_ACTION
    time_stamp
MDAS_AD_DSET_SUMMARY
    (mo_data_id             %together form reference to
    replication_enum        %MDAS_AD_DSET_REPLICATION
    mo_action_id            %references MDAS_TD_ACTION
    spectral_timestamp
    spectral_id             %references MDAS_TD_SPECTRAL_SUMMARY
    spectral_value
    );
```

The MDAS_AD_DSET_AUDIT table is used to log in an audit trail about the usage of the datasets. Every operation performed on every registered dataset is logged in here and can be used to charge users for accessing datasets and also for finding the usage statistics of each replica of the dataset. This information is used to update the *permanence* attribute in the MDAS_AD_DSET_REPLICATION table. If a replica falls below a certain level of permanence, it may be purged from the system or at least moved to an archival storage.

The MDAS_AD_DSET_SUMMARY table is used to store spectral information that is gleaned from the audit trail. That is, for each dataset replica and each operation, one finds the number times the action has been performed in a particular time period (say past minute, past hour, every wednesday in the last year, etc) and are stored with the appropriate *spectral_id*. Spectral information definitions are provided in the MDAS_TD_SPECTRAL_SUMMARY table. The summary information can be used for predicting usage statistics for datasets.

The next two AD tables provide metadata for operations that are borrowed from database

systems but are applied to any dataset in MDAS.

```
MDAS_AD_DSET_AGGREGATION
    (mo_data_id             %references MDAS_CD_DATA
                            % parent data set on which aggregation is being performed
    aggregation_id          %references MDAS_TD_DSET_AGGREGATION
    aggregation_data_id     %references MDAS_CD_DATA
                            % resultant dataset holding the aggregation.
    );
MDAS_AD_DSET_TRIGGER
    (mo_data_id             %together form reference to
    replication_enum        %MDAS_AD_DSET_REPLICATION
    validated_condition
    mo_trigger_id           %references MDAS_TD_DSET_TRIGGER
    );
MDAS_TD_DSET_TRIGGER
    (mo_trigger_id
    trigger_description
    trigger_method          %references MDAS_CD_METHOD
    trigger_mode
    trigger_condition
    destination_data_id
    );
```

Aggregation in DBMS is an operation that is performed on tables to obtain summary information of interest. For example, one may want to find the regional sales totals of a nation-wide department store using the receipts table. In the context of other datasets any derived dataset can be seen as a result of an 'aggregation' operation. In this sense, for each dataset, the MDAS_AD_DSET_AGGREGATION table provides pointers to all derived datasets. This can be seen as an inverse of the lineage data provided by other tables (discussed later). Even though every dataset has a lineage, the aggregation table may contain only pointers to a few important derived datasets. Hence, the information stored in this table will be helpful if a user wants to find a particular aggregate of a dataset; if the aggregation is not available, one may need to recompute it.

Triggers in DBMS are used for enforcing general forms of integrity such as business rules (eg., no salary can be increased by more than 10%) and for performing actions (possibly outside the database) on insert delete or update operations. The MDAS_AD_DSET_TRIGGER table provides metadata to achieve similar capabilities for all types of datasets that are registered with the system. The MDAS_TD_DSET_TRIGGER table provides the *trigger_condition* for the trigger, the *method* that needs to be performed if the condition is true and the time at which it is performed using the *trigger_mode* attribute. The *trigger_mode* may be used to defer the action, perform it immediately or to spawn a new transaction to be done independent of the triggering transaction. It may also be used to suggest whether the trigger is a one-time only trigger or a continuous trigger. The *mode* attribute can also be used to turn on or turn off the trigger. The *validated_condition* can be used to cache parts of conditions that have already been validated by previous trigger actions. With this facility,

one can perform a trigger which has a validation condition that requires information from earlier states of the dataset.

The AGGREGATION and TRIGGER tables are not available for other types of entities in MDAS.

Next we see the metadata support offered by MDAS for finding the lineage of a dataset. By lineage we mean the information about datasets, resources, users and methods that were involved in creating the database. Lineage information does not apply to updates to datasets.

```
MDAS_AD_DSET_LINEAGE_DATA
    (mo_data_id             %together form reference to
    replication_enum        %MDAS_AD_DSET_REPLICATION
    parent_data_id          %references MDAS_CD_DATA
    parent_in_data_enum     );
MDAS_AD_DSET_LINEAGE_METHOD
    (mo_data_id             %together form reference to
    replication_enum        %MDAS_AD_DSET_REPLICATION
    parent_method_id        %references MDAS_CD_METHOD
    child_output_enum
    );
MDAS_AD_DSET_LINEAGE_USER
    (mo_data_id             %together form reference to
    replication_enum        %MDAS_AD_DSET_REPLICATION
    owner_user_id           %references MDAS_CD_USER
    );
MDAS_AD_DSET_LINEAGE_PARAMETER
    (mo_data_id             %together form reference to
    replication_enum        %MDAS_AD_DSET_REPLICATION
    parameter_enum
    parameter_value
    );
MDAS_AD_DSET_LINEAGE_RESOURCE
    (mo_data_id,            %together form reference to
    replication_enum        %MDAS_AD_DSET_REPLICATION
    sub_method_enum
    parent_resource_id      %references MDAS_CD_RESOURCE
    );
```

As explained later, we consider that each method (including compound methods) has an enumerated list of input datasets, an enumerated list of parameters which when applied to the method, produces an enumerated list of output datasets. Each sub-method of a compound method may be performed on a different resource and the method itself may be initiated by more than one user. In providing metadata about the lineage of a dataset, we consider it to be an output dataset emitted by a method, and capture all relevant data that went into its creation. The MDAS_AD_DSET_LINEAGE_DATA table

captures the parent datasets (and their position in the enumerated list of input datasets) that created the dataset of interest given by *mo_data_id, replication_enum*. The method that created the dataset is captured in the table MDAS_AD_DSET_LINEAGE_METHOD along with the information about the position of the created dataset in the output list of datasets. The MDAS_AD_DSET_LINEAGE_USER table captures the user information and the MDAS_AD_DSET_LINEAGE_PARAMETER table records the information about parameter values used in the creation. Finally, the MDAS_AD_DSET_LINEAGE_RESOURCE table provides information about the various resources used at each step in the method invocation.

The MDAS_AD_DSET_ALIAS table given below is a facility for each user (or group) to give an alias name for any dataset, apart from the one provided in the MDAS_CD_DATA table.

MDAS_AD_DSET_ALIAS
(mo_data_id %references MDAS_CD_DATA
mo_data_name
mo_user_id %references MDAS_CD_USER
);

The following tables provide the necessary mapping for using SD metadata tables in the MDAS framework.

MDAS_AD_DSET_SD
(mo_data_id %references MDAS_CD_DATA
sd_description
sd_data_id %references MDAS_CD_DATA
);
MDAS_AD_DSET_TYPE_SD
(mo_data_type %references MDAS_TD_DATA_TYPE
sd_description
sd_data_id %references MDAS_CD_DATA
);

SD meta data tables are defined outside the core metadata tables of MDAS and are useful in storing application-specific metadata. The MDAS system will be able to provide rudimentary metadata for access and computation without the aid of SD tables. The SD tables are useful in describing additional properties of datasets, methods, resources and users. For example, the search indices provided by systems such as Open Text for documents and patents can be useful SD tables that may allow users to access the documents through keywords. Properties of computing platforms such as the speed, number of processors, size of memory, etc., can be useful SD metadata that can be used by a scheduler to find optimal platforms for processes.

The MDAS_AD_DSET_SD table provides entries about SD tables that are relevant to particular datasets, and the MDAS_AD_DSET_TYPE_SD provides the same information for datasets of common types (eg. all patent type datasets may share one or more index SD

tables). The *description* attribute provides keywords or definitions about the contents of the SD table. A query to MDAS can use the *description* value to find appropriate SD tables for a given type of dataset (or particular dataset) and search in it for relevant information. For example, the following SD table (also registered in MDAS) provides keyword-to-patent dataset mapping that can aid in finding relevant datasets of interest.

SD_024
(keyword *% word that occurs in*
mo_data_id *%a (patent) dataset registered in MDAS*
);

The SD_024 table can be made visible to the user through the MDA system by having an entry such as $('patent','keyindex',\#id\_for\_SD\_024)$ in the MDAS_AD_DSET_TYPE_SD table. MDAS will provide facilities to search in such SD metadata tables.

Next we discuss the schema of metadata tables for methods. We do not discuss the AD and TD tables that have similar functionalities with respect to the dataset metadata tables.

MDAS_CD_METHOD
(mo_method_id
mo_method_name
mo_method_type_id
verification_key *% public key for verification*
verification_id *%references MDAS_TD_VERIFICATION*
public_key *% public key for encryption*
public_id *%references MDAS_TD_ENCRYPTION*
is_compound
);

The MDAS_CD_METHOD table is similar to the CD table for datasets, except that it contains an *is_compound* attribute (instead of an *is_partitioned* attribute) which can be used to register compound methods. In MDAS each compound method is considered as a single method with an enumerated list of input datasets, an enumerated list of parameters and an enumerated list of output datasets. The metadata about these enumerated lists are stored in corresponding APPLICATION AD tables discussed next. The mapping of the compound method to its constituent sub-methods and the mapping of the inputs and outputs between the two and the interaction between the sub-methods are captured in the COMPOUND AD tables.

In MDAS we provide two types of metadata for aiding in implementing authentication and encryption mechanisms. We follow the mechanisms outlined in [?] when defining the metadata. We consider that a dataset, user, resource or method may use private signatures to authenticate their validity using privately held *authentication* keys and the receiver of information from these sources will use publicly available *verification* keys to check the validity of their communications. The above mechanism helps in ensuring that the receipients in the communication can be convinced about the originator's validity. The

verification_key in MDAS_CD_METHOD (and in other CD tables) are used for verifying a dataset or message for its authenticity.

The *verification_id* provides a handle to the verification method that needs to be used in the checking process. The MDAS_TD_VERIFICATION table that provides the method and its parameters is given below. A similar table called MDAS_TD_AUTHENTICATION helps in applying the authenticating signature to a dataset or message before communicating them.

MDAS_TD_VERIFICATION
(verification_id
verification_description *% description*
verification_name *% verification scheme (eg. DSS,RSA,MD5,SHA,etc.)*
verification_kind *% kind of signature (eg. symmetric, public)*
verification_keylength
verification_public_info *% other info such as modulo, etc*
verification_method_id % references MDAS_CD_METHOD
);

To ensure secrecy, a dataset, user, resource or method may use the publicly available *encryption* keys of the recievers to encypher the dataset or messages that are being communicated to the receiver. The receiver (a user, resource or method) in turn will use its own privately held *decryption* key to decypher the message. This mechanism helps in ensuring secure communications between domains possibly separated by insecure communication channels. The *encryption_key* in MDAS_CD_METHOD are used for encrypting a dataset. The encrption key is available for other CD tables except that for datasets. Datasets are not designed to receive any information and hence do not require any encryption mechanism.

The *encryption_id* provides a handle to the encryption method that needs to be used in the cypher process. The MDAS_TD_ENCRYPTION table that provides the method and its parameters is given below. A similar table called MDAS_TD_DECRYPTION helps in applying the decyphering a dataset or message after receiving them.

MDAS_TD_ENCRYPTION
(encryption_id
encryption_description *% description*
encryption_name *% encryption scheme*
encryption_kind *% kind of encryption (eg. symmetric, public)*
encryption_keylength
encryption_public_info *% other info such as modulo, etc*
encryption_method_id % references MDAS_CD_METHOD
):

Apart from the publicly available signature verification keys and encryption keys, their counterparts of private authentication and decryptio keys are stored in (or at leaset via) MDAS in corresponding AD tables. Below, we show the AD tables for methods. Similar tables are available for resources, users and datasets.

MDAS_AD_METH_AUTHENTICATION_KEY

(mo_method_id	*% references MDAS_CD_METHOD*
private_key	*% public key is stored in a secure DBMS or*
private_lockbox_method	*% a method is used to access*
	% a private key held in a lock box
	% references MDAS_CD_METHOD
authentication_id	*%references MDAS_TD_AUTHENTICATION*
);	

MDAS_AD_METH_DECRYPTION_KEY

(mo_method_id	*% references MDAS_CD_METHOD*
private_key	
private_lockbox_method	*% references MDAS_CD_METHOD*
decryption_id	*%references MDAS_TD_DECRYPTION*
);	

We assume that either a private key is held in a secure DBMS or a method is available for getting hold of a key from a software lock box. The *authentication_id* (sim. decryption_id) provide a handle to the methods that is used to apply the authentication key (or decryption key) to a message or dataset.

MDAS_AD_METH_APPLICATION_PARAMETER

(mo_method_id	*%references MDAS_CD_METHOD*
parameter_enum	
parameter_type	
parameter_name	
parameter_default_value	
);	

MDAS_AD_METH_APPLICATION_OUTPUT

(mo_method_id	*%references MDAS_CD_METHOD*
out_enum	
out_data_type_id	
out_data_name	
out_default_value	
);	

MDAS_AD_METH_APPLICATION_INPUT

(mo_method_id	*%references MDAS_CD_METHOD*
in_enum	
in_data_type_id	
in_data_name	
in_default_value	
);	

MDAS_AD_METH_APPLICATION_REQUIREMENTS

(mo_method_id	*%references MDAS_CD_METHOD*
misc_name	
misc_type	
misc_value	
);	

MDAS_AD_METH_APPLICATION_PREDICTION
(mo_method_id *%references MDAS_CD_METHOD*
replication_enum
prediction_method_id *%references MDAS_CD_METHOD*
prediction_description
);

The APPLICATION table captures metadata about how to use the methods registered with MDAS. The MDAS_AD_METH_APPLICATION_PARAMETER table contains information about all the command-line parameters that can be given when invoking the method. The parameters are enumerated in a list and this enumeration is used when using the method as a sub-method in a compound method or to record lineage information as discussed for datasets. The name and type of the parameter is included along with a default value that can be used in case the user fails to supply a value.

The MDAS_AD_METH_APPLICATION_OUTPUT table provides information about the output datasets created by the method. The type of dataset is also noted along with the name that is used to describe the output dataset. A default value for naming the output dataset is also provided in case the user fails to give a name for storing the output. The MDAS_AD_METH_APPLICATION_INPUT table provides similar information about input datasets to a method. Default input datasets are used in case user has not assigned a value. This will be helpful in cases where some of the inputs (eg. a dictionary) can be understood by default.

The MDAS_AD_METH_APPLICATION_REQUIREMENTS table captures all other information that may be relevant and necessary when invoking the method. For example, a method may require a large memory space in order to execute and this requirement may be logged in this table. Hence, any software, hardware, protocol requirements may be given as metadata in this table. The MDAS_AD_METH_APPLICATION_PREDICTION provides metadata to predict the performance of a method. The *prediction_method_id* can take parameters such as sizes of input and provide a prediction about the time required by the method to complete. This information could be used by a scheduler for efficiently using resources or to reserve resources for later execution (see discussion of the MDAS_AD_DSET_LOCK table earlier.)

MDAS_AD_METH_COMPOUND_METHOD_MAP
(mo_method_id *%references MDAS_CD_METHOD*
method_enum
sub_method_id *%references MDAS_CD_METHOD*
);
MDAS_AD_METH_COMPOUND_PARAMETER_MAP
(mo_method_id *%references MDAS_CD_METHOD*
method_param_enum
sub_method_id *%references MDAS_CD_METHOD*
sub_method_param_enum
);
MDAS_AD_METH_COMPOUND_DSET_MAP

(mo_method_id *%references MDAS_CD_METHOD*
producer_method_enum
producer_out_enum
consumer_method_enum
consumer_in_enum
);

The compound AD tables shown above provide the mapping between a compound method to its constituent sub-methods. The MDAS_AD_METH_COMPOUND_METHOD_MAP table stores the method-to-methods map and also shows an enumeration of the sub-method in the order that they need to be applied to obtain the functionality of the compound method. For example, assume that a compound method *cl* consist of *compile* and *link* sub-methods to be performed in that order, then the two methods are logged in the table with enumeration 1 and 2.

The MDAS_AD_METH_COMPOUND_PARAMETER_MAP table maps the parameters given for the compound method to parameters required by the sub-methods. Again considering the compound method *cl*, it will have all parameters needed by its sub-methods stored in MDAS_AD_METH_COMPOUND_PARAMETER_MAP. Some parameters of *cl* may be required by both sub-methods also.

The MDAS_AD_METH_COMPOUND_DSET_MAP table captures three types of mappings: from the compound method input datasets to the input datasets of the sub-methods, from the output datasets of the sub-methods to the output datasets of the compound method and, the interconnection between two submethods, where one of the sub-method's output may be the input of another sub-method. We consider that the compound method is given an enumeration number 0 and the sub-methods are numbered from 1 onwards. The compound method is considered to be the *producer* of input datasets and *consumer* of output datasets. Note that a dataset (either input dataset or a dataset that is output by a sub-method) can be used as input by more than one sub-method. For example, for the data flow diagram given in Figure 4.5 appropriate mappings are captured in the sets of tuples shown in Tables 4.14 and 4.15; we assume that the compound method has 24 as its identifier. Tables 4.14 describes the dataset flow between the sub-methods of a compound method as well as the usage of input datasets and creation of output datasets by the the sub-methods. Tables 4.15 describes the usage of parameters by the sub-methods. In the data flow diagram in Figure 4.5 a few datasets (output files) are created by the sub-methods which are not used by other sub-methods or by given as output datasets. These datasets are scratch files produced by these methods. In MDAS, we do not keep track of internal datasets created by the sub-methods, only the datasets that are explicitly outputted are registered. If one needs to know the intermediate datasets, then one can obtain them by executing the compound method and explicitly outputting the required datasets.

The meta information in the MDAS_AD_METH_COMPOUND_DSET_MAP table can be used by an intelligent scheduler to parallelize a compound method. For example, in the above method sub-methods **1** and **2** can be performed in parallel.

MDAS_AD_CONVERT_METHOD_ID
(mo_method_id *%references MDAS_CD_METHOD*

mo_method_id	*producer_method_enum*	*producer_out_enum*	*consumer_method_enum*	*consumer_in_enum*
24	0	1	1	1
24	0	1	2	1
24	0	2	2	2
24	1	1	4	1
24	1	2	3	1
24	2	1	3	2
24	3	1	4	2
24	3	1	0	2
24	4	1	0	1

Table 4.14: Sample MDAS_AD_METH_COMPOUND_DSET_MAP table

mo_method_id	*method_param_enum*	*sub_method_id*	*sub_method_param_enum*
24	1	1	1
24	2	1	2
24	2	3	1
24	3	4	1

Table 4.15: Sample MDAS_AD_METH_COMPOUND_PARAMETER_MAP table

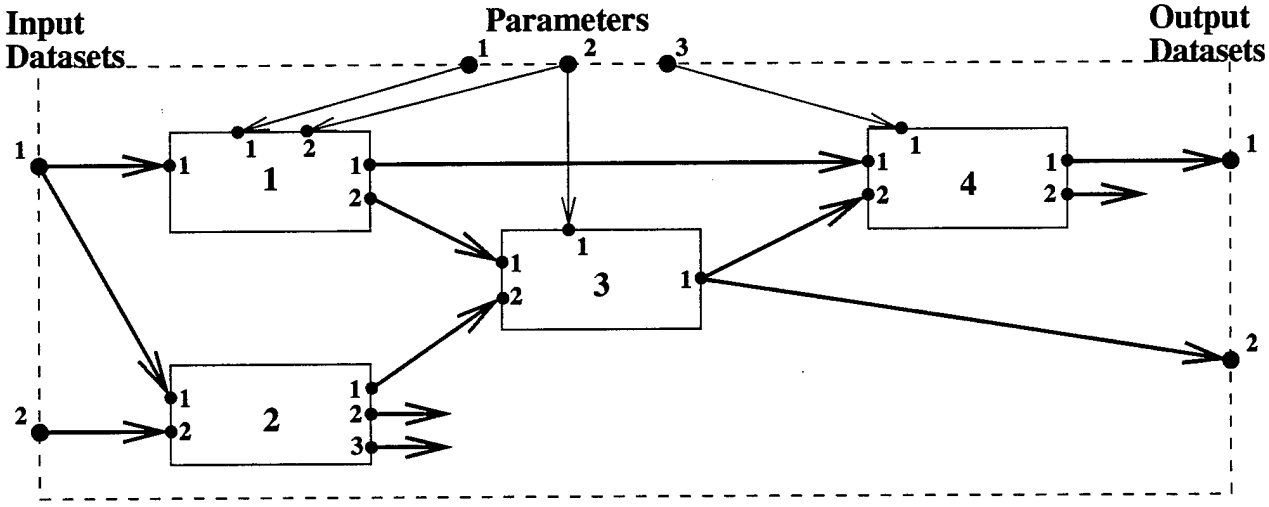

Figure 4.5: Example: Data flow in a Compound Method

mo_indata_type
mo_outdata_type
);
MDAS_AD_CONVERT_METHOD_TYPE
(mo_method_type
mo_indata_type
mo_outdata_type
);

The conversion tables given above provide a means of finding methods that are applicable for conversion of a dataset from one type to another. An important aspect of MDAS is to provide *format transparency.* Hence, if a user wants to print a file that is not in *latex* format to a Postscript printer and no replica of an equivalent file in *postscript* format exists, then MDAS will use the MDAS_AD_CONVERT_METHOD_ID table to find appropriate methods that can perform the format transformation before feeding the file to the printer. The MDAS_AD_CONVERT_METHOD_TYPE file stores metadata about format conversion, but gives a class of methods that can perform the transformation instead of individual methods.

Next we discuss the schema of metadata tables for resources. Again, we do not discuss the AD and TD tables that have been discussed for other types of MDAS elements.

MDAS_CD_RESOURCE
(mo_resource_id
mo_resource_name
mo_resource_type_id
global_authentication_id
);

The CD table has similar functionality as the CD tables of datasets and methods. Note that the concept of a resource in MDAS is very diverse and includes hardware resources such as storage systems, peripheral systems, memory systems (such as cache), communication systems, computing systems, etc., and software resources such as operating systems, file system, archival system, database systems, scheduling systems, MDAS systems, digital library systems, etc. The properties of each special types of resources are captured in non-system SD metadata tables.

MDAS_AD_FUNCTION_ON_RESOURCE
(mo_resource_id *%references MDAS_CD_RESOURCE*
mo_method_id
function_id
);
MDAS_AD_RSRC_APPLICATION_PREDICTION
(mo_resource_id *%references MDAS_CD_RESOURCE*
replication_enum
prediction_method_id
prediction_description

);

The MDAS_AD_FUNCTION_ON_RESOURCE table provides metadata about the functions that can be applied to each resource. For example, an archival system may have functions such as *open, close, create, read, write, seek*, etc. Such functions are registered in MDAS through this table using the *function_id* attribute. The corresponding TD table, MDAS_TD_RSRC_FUNCTION, provides attributes for storing the name, a description of the function and also whether it complies to any standards (eg. functions to a DBMS may be ODBC compliant, or a parallel executable function may be MPI compliant). Moreover, the *mo_method_id* in MDAS_AD_FUNCTION_ON_RESOURCE table provides the method that needs to be invoked to obtain the functionality. This provides *method transparency* when applying the functions which can be generic for resources of the same type.

The MDAS_AD_RSRC_APPLICATION_PREDICTION table is used to store data that can be used to predict the performance of each resource. We consider that with proper parameters such as size of datasets, function being applied, etc., one can predict performance using the associated *prediction method.* More than one prediction method can be associated with each resource.

Finally, we discuss the schema of metadata tables for users. We discuss the CD table and group AD metadata table only as other tables are similar in semantics with corresponding tables discussed earlier.

MDAS_CD_USER
(mo_user_id
mo_user_name
mo_user_address
mo_user_phone
mo_user_phone2
mo_user_fax
mo_user_email
mo_user_domain
mo_user_type_id
authentication_id
);
MDAS_AD_USER_GROUP
(mo_user_id *%references MDAS_CD_USER*
group_user_id *%references MDAS_CD_USER*
);

The MDAS_CD_USER contains information about users and groups. The user type can be used to define group hierarchies. The semantics of authentication is similar to those of other elemental types in MDAS. The MDAS_AD_USER_GROUP table stores group membership information.

4.9.6 Implementation Issues

The previous section demonstrated the important role played by metadata in the MDAS architecture and also provided a schema for the MDAS Metadata Catalog. This section provides a discussion of issues related to creating and maintaining this metadata.

4.9.6.1 Generation

Some of the metadata recorded in the catalogs is provided directly by the user, while other metadata can be obtained automatically by pre-processing the input (eg. data sets and methods), actively searching for metainformation (eg. resource discovery) or, monitoring the progress of an operation (eg. resource or process metadata). The system includes mechanisms to capture all of these types of metadata.

4.9.6.2 Representation

Addressing issues related to metadata representation is critical to the goal of providing seamless integration of metadata across distributed heterogenous systems. A metadata representation scheme must provide the following:

1. Convenient methods for inserting, updating and viewing metadata.
2. Ease of transportation of metadata among diverse distributed computing and storage environments.
3. Easy translation into various in-memory formats.
4. An inheritance hierarchy. For example, one can define metadata about printers in general, followed by those for laser-printers and color-matrix printers, etc, followed by metadata about specific printer-types (eg. Epson II stylus printer) and finally metadata parameters (eg. address, buffer size etc) about a specific printer attached to a LAN or workstation.

The above representational issues will be handled by defining a language for representing metadata, called the *MetaData Markup Language (MDML)*. This language will provide a common distribution format for portability, and a user-friendly scheme for entering and updating metadata. The common format will also allow one to write translation methods for storage and memory representation at language-specific and system-specific levels. Moreover, one can also define a set of methods that can be applied to metadata that can be used for creating, maintaining and querying purposes.

4.9.6.3 Storage

Since the quantity of metadata can overwhelm any system, it is important to consider issues related to storage hierarchies, local versus remote location of metadata, distribution and

fragmentation of data, and replication. An important aspect of metadata management is the ability to store and access data based on usage patterns, archival requirements, and access time requirements. The system must provide the application an interface to the metadata which is independent of where the data is stored and how it is managed.

4.9.6.4 Maintenance

Important issues in metadata maintenance are fault-tolerance, schema evolution, and versioning. MDAS catalog services must be highly available and fault tolerant to ensure that applications are not unduly disconnected from resources. Schema evolution and versioning must be supported since database schemas tend to evolve over time and, at any given instant, it is likely that different resources available to MDAS are using different versions of the schema. Since the metadata schema and the metadata will evolve over time, considerations for extensibility should be central to the design.

4.9.6.5 Retrieval

The system must support efficient retrieval of metadata. Since the metadata catalog can reside on different platforms, possibly fragmented and replicated, it is neccesary to provide a fully-transparent retrieval system that can handle issues such as optimization, caching, sharing, authentication and security. It may also be neccesary to provide *active* mechanisms which provide support for triggering an action when a change occurs to a specific piece of metadata.

4.9.6.6 Legacy and Domain-dependent Metadata

Finally, since there already exist vast amounts of domain-specific metadata, defined by various groups of users (eg. biologists) one needs to give careful thought to the design of the system in order to allow assimilation of existing metadata in the new framework. We plan to tackle this problem by providing automated means for encoding domain-specific metadata in a uniform manner and also by porting existing metadata into the MDAS framework.

4.10 Library and Catalog Table Bindings

TBD.

4.11 Low-Level Interface

This section discusses the MDAS Low-Level interface in detail. Two functionalities are supported here: *drivers* and *internals*. Nothing at this level is for use by application programs. Datatypes unique to this level are discussed in section 4.11.1. Function prototypes for MDAS drivers and internals are presented in sections 4.11.2 and 4.11.3. Examples concerning driver implementations are given in section **??**.

4.11.1 Low-Level Data Types

TBD.

4.11.1.1 MDAS_INIT_STRUCT

TBD.

4.11.1.2 MDAS_TICKET_STRUCT

TBD.

4.11.1.3 MDAS_DRIVER_STRUCT

TBD.

4.11.1.4 MDAS_RSRC_STRUCT

TBD.

4.11.1.5 MDAS_RSRCC_STRUCT

TBD.

4.11.1.6 MDAS_LLIST_STRUCT

TBD.

Prototype Name	Implemented Name	
	Fortran 90	C, C++
MDAS_*_CONN	mdasF_db2_conn	mdasC_db2_conn
MDAS_*_DCON	mdasF_db2_dcon	mdasC_db2_dcon
MDAS_*_GET	mdasF_db2_get	mdasC_db2_get
MDAS_*_PUT	mdasF_db2_put	mdasC_db2_put
MDAS_*_OPEN	mdasF_db2_open	mdasC_db2_open
MDAS_*_CLOSE	mdasF_db2_close	mdasC_db2_close
MDAS_*_READ_BLK	mdasF_db2_read_blk	mdasC_db2_read_blk
MDAS_*_WRITE_BLK	mdasF_db2_write_blk	mdasC_db2_write_blk
MDAS_*_EXEC_TYPES	mdasF_db2_exec_types	mdasC_db2_exec_types
MDAS_*_EXEC	mdasF_db2_exec	mdasC_db2_exec
MDAS_*_CAT_EXISTS	mdasF_db2_cat_exists	mdasC_db2_cat_exists
MDAS_*_CAT_MAKE	mdasF_db2_cat_make	mdasC_db2_cat_make
MDAS_*_CAT_DEL	mdasF_db2_cat_del	mdasC_db2_cat_del
MDAS_*_INFO_INQUIRE	mdasF_db2_info_inquire	mdasC_db2_info_inquire
MDAS_*_INFO_REGISTER	mdasF_db2_info_register	mdasC_db2_info_register
MDAS_*_INFO_UPDATE	mdasF_db2_info_update	mdasC_db2_info_update
MDAS_*_INFO_PURGE	mdasF_db2_info_purge	mdasC_db2_info_purge

Table 4.16: Driver function naming conventions for driver "db2"

4.11.2 Low-Level Drivers

At the lowest level, MDAS maintains a set of architecture and resource specific drivers. These are for internal use of the MDAS-library Mid-Level engines and daemons, and not exposed to application programs.

Currently, each driver contains about 30 routines. Depending on the nature of the function and type of resource, these routines range from low to medium level of code complexity. The argument lists for each set of driver routines are identical.

All implementations must implement a driver for the operating system to be used in the run-time environment. Built-in rules in the MDAS Build Environment enforce this policy.

4.11.2.1 Naming Conventions

All Low-Level prototypes have the prefix `MDAS_*_` where * is the driver name. Driver names and the availability of drivers is implementation dependent. A translation table of prototype names and their actual names for a "db2" driver is given in table 4.16.

4.11.2.2 Built-In Drivers

Discussion: TBD.

4.11.2.3 Access

```
MDAS_*_CONN(user, ticket, server, comm, servh, status)
     user:   (IN)      MDAS_INFOH
     ticket: (IN)      MDAS_INFOH
     server: (IN)      MDAS_INFOH
     comm:   (IN)      MDAS_handle
     servh:  (OUT)     MDAS_SERVH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	connection established
MDAS_ERR_COMM	error	communicator operation failed
MDAS_ERR_MPI	error	severe MPI error
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_TICKET	error	access error
MDAS_ERR_SERVH	error	server not available

`MDAS_*_CONN()` connects the user named in **user** to the service described in **server** with authorization protocol and key specified in **ticket**. The **server** argument is guaranteed to describe 1 unique service. Likewise, **user** and **ticket** are guaranteed to contain the description of 1 item.

If **comm** is non-NULL, it is guaranteed to be a valid communicator and `MDAS_*_CONN()` may execute collective operation on **comm**. If **comm** is used to establish an MPI *intercommunicator* with a remote service then this new handle must be cached in **servh**. If the driver does not use MPI protocols, then **comm** can be ignored.

The returned **servh** handle maintains the functionality of a physical connection across time-outs. The actual implementation may be that in the event of a low-level operation on a time-out, the connection is re-instated by the call responsible for the operation.

```
MDAS_*_DCON(servh, comm, status)
     servh:  (IN)      MDAS_SERVH
     comm:   (IN)      MDAS_handle
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	server deallocated
MDAS_ERR_COMM	error	communicator operation failed
MDAS_ERR_MPI	error	severe MPI error
MDAS_ERR_MEMORY	error	unable to deallocate memory
MDAS_ERR_SERVER	error	server not available

MDAS_*_DCON() closes the connection specified by servh and frees any memory associated with the structure. If MDAS_*_CONN() was used to establish an intercommunicator, it should be released.

4.11.2.4 Cache Operations

4.11.2.4.1 MDAS_*_GET()

```
MDAS_*_GET(dset, dstkt, servh, comm, cache, status)
     dset:   (IN)      MDAS_INFOH
     dstkt:  (IN)      MDAS_INFOH
     servh:  (IN)      MDAS_SERVH
     comm:   (IN)      MDAS_handle
     cache:  (IN)      MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

MDAS_*_GET() discussion: TBD.

4.11.2.4.2 MDAS_*_PUT()

```
MDAS_*_PUT(cache, dset, dstkt, servh, comm, status)
     cache:  (IN)      MDAS_INFOH
     dset:   (IN)      MDAS_INFOH
     dstkt:  (IN)      MDAS_INFOH
     servh:  (IN)      MDAS_SERVH
     comm:   (IN)      MDAS_handle
     status: (IN/OUT)  MDAS_status
```

MDAS_*_PUT() discussion: TBD.

4.11.2.5 Data Handles

```
MDAS_*_OPEN(dset, dstkt, servh, comm, mode, dh, status)
     dset:   (IN)      MDAS_INFOH
     mode:   (IN)      MDAS_INFOH
```

```
dstkt:  (IN)      MDAS_INFOH
servh:  (IN)      MDAS_SERVH
comm:   (IN)      MDAS_handle
dh:     (OUT)     MDAS_DATAH
status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	data set open on **dh**
MDAS_ERR_COMM	error	communicator operation failed
MDAS_ERR_MPI	error	severe MPI error
MDAS_ERR_TICKET	error	invalid ticket
MDAS_ERR_MODE	error	invalid **mode**
MDAS_ERR_READ	error	read access denied
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_SERVER	error	server not available
MDAS_ERR_DATASET	error	data set not available

MDAS_*_OPEN() opens a generalized data handle for data set **dset** managed by the service **servh**. The **mode** argument specifies the desired I/O mode. Any authentication ticket required for the data set is provided in **dstkt**. The handle **servh** is guaranteed to be valid. If **comm** is non-NULL, it is guaranteed to be valid and MDAS_*_OPEN() may perform collective operations. If an MPI *intercommunicator* is established to perform the I/O operation, it should be cached in **dh**. An error is returned with a NULL **dh** on failure.

```
MDAS_*_CLOSE(dh, comm, status)
     dh:     (IN/OUT)  MDAS_DATAH
     comm:   (IN)      MDAS_handle
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	**dh** deallocated
MDAS_ERR_COMM	error	communicator operation failed
MDAS_ERR_MPI	error	severe MPI error
MDAS_ERR_MEMORY	error	unable to deallocate memory
MDAS_ERR_SERVER	error	server not available
MDAS_ERR_IO	error	I/O error on close

MDAS_*_CLOSE() closes the data stream, etc. specified by **dh** (and possibly **comm**) and frees any memory associated with the structure.

4.11.2.6 Block I/O

The MDAS drivers provide a basic block I/O facility for reading and writing bytes on open data handles.

```
MDAS_*_READ_BLK(dh, count, buffer, status)
     dh:     (IN/OUT)  MDAS_DATAH
     count:  (IN)      MDAS_integer
     buffer: (IN/OUT)  MDAS_handle
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	bytes read
MDAS_WARN_EOS	error	found logical end of stream
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_SERVER	error	server not available
MDAS_ERR_DATASET	error	data set not available
MDAS_ERR_IO	error	I/O operation failed

MDAS_*_READ_BLK() reads **count** bytes from **dh** and places them in **buffer**. A warning is returned in **status** if the end-of-file, end-of-stream, etc. is reached.

```
MDAS_*_WRITE_BLK(count, buffer, dh, status)
     count:  (IN)      MDAS_integer
     buffer: (IN/OUT)  MDAS_handle
     dh:     (IN/OUT)  MDAS_DATAH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	bytes read
MDAS_ERR_EOS	error	found physical end of stream
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_SERVER	error	server not available
MDAS_ERR_DATASET	error	data set not available
MDAS_ERR_IO	error	I/O operation failed

MDAS_*_WRITE_BLK() writes **count** bytes from **buffer** to **dh**. A error is returned in **status** if the physical end-of-file, end-of-stream, etc. is reached.

4.11.2.7 Executing Methods

4.11.2.7.1 MDAS_*_EXEC_TYPES()

```
MDAS_*_EXEC_TYPES(server, types, status)
     server: (IN)      MDAS_INFOH
     types:  (OUT)     MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

MDAS_*_EXEC_TYPES discussion: TBD.

4.11.2.7.2 MDAS_*_EXEC()

```
MDAS_*_EXEC(method, params, server, ds_in, ds_out, status)
     method: (IN/OUT)  MDAS_INFOH
     params: (IN/OUT)  MDAS_INFOH
     server: (IN/OUT)  MDAS_INFOH
     ds_in:  (IN/OUT)  MDAS_INFOH
     ds_out: (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

MDAS_*_EXEC discussion: TBD.

4.11.2.8 Catalog Operations

```
MDAS_*_CAT_EXISTS(name, servh, status)
     name:   (IN)      MDAS_string
     servh:  (IN)      MDAS_SERVH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	catalog exists
MDAS_ERR_SERVER	error	server not available
MDAS_ERR_CATALOG	error	catalog not available

MDAS_*_CAT_EXISTS() checks for the existance of an MDAS catalog named in name on servh. The caller may check if the driver supports catalog functions by calling MDAS_*_CAT_EXISTS() with NULL for both name and servh.

```
MDAS_*_CAT_MAKE(name, servh, status)
     name:   (IN)      MDAS_string
     servh:  (IN)      MDAS_SERVH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	catalog created
MDAS_ERR_SERVER	error	server not available
MDAS_ERR_CATALOG	error	catalog broken

MDAS_*_CAT_MAKE() creates an MDAS catalog named in name on servh.

```
MDAS_*_CAT_DEL(name, servh, status)
     name:   (IN)      MDAS_string
     servh:  (IN)      MDAS_SERVH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	catalog deleted
MDAS_ERR_SERVER	error	server not available
MDAS_ERR_CATALOG	error	catalog broken

MDAS_*_CAT_DEL() deletes the MDAS catalog named in **name** on **servh**.

4.11.2.9 Catalog Info Operations

```
MDAS_*_INFO_INQUIRE(servh, info, result, status)
     servh:  (IN)      MDAS_SERVH
     info:   (IN)      MDAS_INFOH
     result: (OUT)     MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	result(s) obtained
MDAS_WARN_RESULT	warning	**result** is empty
MDAS_WARN_SPOOL	warning	**result** spooled
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_READ	error	read access denied
MDAS_ERR_RESULT	error	unable to return **result**
MDAS_ERR_SERVH	error	**server** not available

The catalog server connection **servh** is guaranteed to be valid. The **Info** argument **info** is also guaranteed to be valid. The conditions and criteria argument may be NULL. A valid handle is supplied to **result**, in which any results are returned. The result may be spooled if the magnitude exceeds the driver's capability. The result must be spooled if the MDAS_SPOOL directive (see MDAS_DIRECTIVE) is set in **cond**. If the result is spooled, the MDAS_WARN_SPOOL status bit is set.

```
MDAS_*_INFO_REGISTER(servh, info, status)
     servh:  (IN)      MDAS_SERVH
     info:   (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	**info** registered
MDAS_WARN_REGISTER	warning	**info** previously registered
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_SERVH	error	**server** not available

MDAS_*_INFO_REGISTER() registers Info for one MDAS_TYPE. The catalog server connection servh is guaranteed to be valid. The info argument is also guaranteed to be valid and contain at least the minimal amount of required metadata for catalog registration. The catalog MDAS_ID of the registered Info is returned in info. If the info is spooled, the MDAS_WARN_SPOOL status bit is set.

```
MDAS_*_INFO_UPDATE(servh, info, status)
     servh:  (IN)      MDAS_SERVH
     info:   (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	info updated
MDAS_ERR_MEMORY	error	unable to allocate memory
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_REGISTER	error	invalid MDAS_ID
MDAS_ERR_SERVH	error	server not available

MDAS_*_INFO_UPDATE() adds and/or changes catalog Info for one MDAS_ID. The catalog server connection servh is guaranteed to be valid. The info argument is also guaranteed to contain one MDAS_ID and at least one attribute to be added or changed. All attributes in info are assumed to be additions or changes. If the info is spooled, the MDAS_WARN_SPOOL status bit is set.

```
MDAS_*_INFO_PURGE(servh, info, status)
     servh:  (IN)      MDAS_SERVH
     info:   (IN/OUT)  MDAS_INFOH
     status: (IN/OUT)  MDAS_status
```

status codes	type	meaning
MDAS_SUCCESS	success	info purged
MDAS_ERR_WRITE	error	write access denied
MDAS_ERR_REGISTER	error	invalid MDAS_ID
MDAS_ERR_SERVH	error	server not available

MDAS_*_INFO_PURGE() deletes one or more MDAS_ID's and their associated Info in an MDAS catalog. The catalog server connection servh is guaranteed to be valid. The info argument is also guaranteed to contain at least one MDAS_ID and no other attributes. All id#'s in info should be purged. If the info is spooled, the MDAS_WARN_SPOOL status bit is set. If one or more MDAS_ID's in info are invalid, the reponsible id#'s are returned in info and the MDAS_ERR_REGISTER status bit is set.

4.11.3 MDAS Internals

TBD.

4.11.3.1 Parameters and Macros

TBD.

Need to discuss how parameter and macro files are built automatically, what the parameters and macros are, etc.

4.11.3.2 Global Variables

TBD.

4.11.3.2.1 `MDAS_VERBOSE`

TBD.

4.11.3.3 Driver Cross-Reference Tables

TBD.

4.11.3.4 Prototypes

TBD.

`MDAS_InitStructCtor()`

`MDAS_InitStructDtor()`

`MDAS_args()`

`MDAS_env()`

`MDAS_gethome()`

`MDAS_rc2home()`

`MDAS_rc2dflt()`

`MDAS_validhome()`

`MDAS_getrc()`

`MDAS_validrc()`

`MDAS_gettickets()`

MDAS_rc2tickets()

MDAS_validtickets()

MDAS_TicketCtor()

MDAS_TicketDtor()

MDAS_DriverCtor()

MDAS_DriverDtor()

MDAS_RsrcCtor()

MDAS_RsrcListCtor()

MDAS_PurgeRsrcList()

MDAS_RsrcDtor()

MDAS_InitRsrcC()

MDAS_RsrcCCtor()

MDAS_RsrcCDtor()

MDAS_ExtendRsrcs()

MDAS_ExtendTckts()

MDAS_CmpRsrcs()

MDAS_CmpTckts()

MDAS_TcktListCtor()

MDAS_PurgeTcktList()

MDAS_AddRsrcList()

MDAS_AddTcktList()

MDAS_RsrcCCache()

MDAS_RsrcCPrint()

MDAS_GetVal()

MDAS_InhaleTxt()

MDAS_LListCtor()

MDAS_LListDtor()

4.12 Demonstration and Test Programs

(Under construction.)

4.12.1 MDAS_INIT() Test

```
/* testinit.c */

#include "COPYRIGHT.h"
#include "mdasC.h"

int
main( int argc, char* argv[] )
{
mdasC_status status ;

mdasC_init( argc, argv, NULL, status ) ;
if (status[0] < 0)
{
fprintf(stderr, "mdasC_example : exiting on mdasC_init error\n") ;
exit(1) ;
}
if (status[0] > 0)
{
fprintf(stderr, "mdasC_example : exiting on mdasC_init warning\n") ;
exit(0) ;
}

mdasC_finalize( NULL, status ) ;

exit(0) ;

}
```

4.12.2 "View a Patent" Demo

```
/* viewpatent.c */
#include "mdasC.h"

int main( int argc, char *argv[] )
{
/* feel free to add comments to this program! */
mdasC_infoh patinfo ;
mdasC_infoh patsrcs ;
mdasC_infoh viewinfo ;
```

```
mdasC_infoh viewrsrcs ;
mdasC_status status ;

/* initialize the MDAS library */
mdasC_init(argc, argv, NULL, status) ;

/* describe the data set as we know it */
mdasC_info_create(MDAS_DATASET, patinfo, status) ;
mdasC_info_set_attr(MDAS_NAME, "10001", patinfo, status) ;
mdasC_info_set_attr(MDAS_STOR_GRPN, "patent", patinfo, status) ;


/* describe the display resources as we know them */
mdasC_info_create(MDAS_RESOURCE, viewinfo, status) ;
mdasC_info_record_add(MDAS_RSRC_TYPE, MDAS_DISPLAY,
viewinfo, status) ;
/* This next record would be taken as a default.  Here
   we include it for completeness. */
mdasC_info_record_add(MDAS_RSRC_LOCT, MDAS_LOCAL,
viewinfo, status) ;

/* get the catalog entries that match the patent description */
mdasC_inquire(patinfo, NULL, NULL, patsrcs, status) ;

/* get the catalog entries that match display description */
mdasC_inquire(viewinfo, NULL, NULL, viewrsrcs, status) ;

/* Pipe the patent to the display method.  PIPE will first
   look for matches between formats of patent replicates
   and input formats of display resources and select
   one (data set, resource) pair.  Next, PIPE will call
   MDAS_GET() to bring the data set to the local resource.
   Finally, pipe calls MDAS_EXEC() to have the local copy
   of the patent (returned from GET in "cacheinfo") displayed.
*/
mdasC_pipe(patsrcs, viewrsrcs, status) ;

/* O.K., we're done. */
mdasC_finalize(NULL, status) ;

exit(0) ;
}
```

4.13 Build Environment

Intro ...TBD.

4.13.1 Directory Structure

TBD.

```
MDAS/
     Makefile
     bin/
     config/
     doc/
     src/
```

4.13.2 Build Directories

MDAS creates an architecture-dependent Makefile and binaries from common source tree. For example, a software build on an IBM AIX system might create the following `MDAS-59.AIX.4.1` infrastructure under the main `MDAS` directory tree:

```
MDAS/
     Makefile
     bin/
     config/
     doc/
     src/
          include/
          mdas/
          db2/
          hpss/
          http/
          illustra/
     MDAS-59.AIX.4.1/
          bin/
          lib/
          src/
               include/
                    ../src/include/*
                    mdas_drvr.h
                    mdas_drvr.F
               mdas -> ../src/mdas
               db2 -> ../src/db2
               hpss -> ../src/hpss
```

Compile-time dependencies are determined by architecture resource files in the `MDAS/config/` directory. Include files named `mdas_drvr.*` with contents specific to the available languages and resources are automatically generated in the target `include/` directory. At run-time, parameters and an array of function pointers defined in the `mdas_drvr.*` include files are used to create the linkage between the API and compiled driver functions.

4.13.2.1 Configuration Files

TBD.

4.13.3 Source Development

TBD.

Need to discuss compile-time configuration of drivers.

4.13.3.1 API

TBD.

4.13.3.2 Mid-Level

TBD.

4.13.3.3 Low-Level

TBD.

4.13.3.3.1 Drivers

TBD.

4.13.3.3.2 Internals

TBD.

4.13.4 Automatically Generated Files

TBD.

4.14 Executable Tools

Executable versions of MDAS Library routines and miscellaneous tools to assist with the task of Catalog metadata registration are discussed here.

4.14.1 Catalog Registration

(TBD).

One tacit assumption so far has been that most MDAS library calls have a command line version for use in shell (script) run-time environments.

4.14.2 Catalog Registration

(TBD).

There was a write-up on this in the old MDAS research pages on the web.

The idea is that there were active and passive "tools" to assist with the registration of what we now call Catalog metadata.

The active tools are a set of scripts and/or executables to be used for batch registration of data sets. These would be particularly useful for registering large batches of files on delivered on tape, etc. Another set of active tools exist in the library – primarly for building interfaces for manual interactive registration. For example, the registration of a user by a system administrator.

The passive tools are library internals which attempt to automatically build the minimal set of metadata required to "register" an MDAS entity. These tools are invoked passively as the library encounters (or suspects the presence of) "new" entities. For example, calling `MDAS_GET/PUT/CONNECT/EXEC` invokes the registration procedures. An advantage of this approach is that the metadata for an entity can be grown with use; i.e., the system "learns" about attributes of an entity over time. Caveat: it must start with a minimal set of attributes.

4.15 Agents and Brokers

(TBD).

Optional MDAS agents and service brokers are discussed here.

4.15.1 The MDAS Outreach Program

(TBD).

Chaitan has introduced a hybrid approach which involves an "active" agent "passively" registering entities it discovers on some extent of resources. It would seem that there are two functionalities required for a robust discovery mechanism: a "browser" and a "validator". The former is as Chaitan describes, the latter is a quality assurance tool that checks whether a registered entity still resides and/or functions as described in the catalog. Since MDAS users may want only one of these functions, it would make sense to have two agents (daemons): `mdas_browse` and `mdas_validate`.

This discussion sounds like a paper for Agents '97 (past due) or an ACM Transactions journal.

Once we have MDAS in place, a concern is how we get neccesary up-to-date metadata into the system. While people may be willing to have MDAS run on their systems (for some presumed advantage that they may gain by doing this), they may not be equally willing to supply all the metadata we would like to have, in the first place, and to keep this updated. The reason being that metadata collection could be laborious and low on people's priority of things to do. The suggestion is to think of MDAS metadata discovery "agents" which go about looking for metadata of interest to MDAS and update MDAS catalogs with this information.

There are, obviously, many details to ponder but, in essence, we create agents for each type of resource and set agents loose on the systems on which MDAS is running or to which it has access. Depending on how friendly a system (or sys admin) wants to be to MDAS, there will be various levels at which these agents could be run, e.g. user level process, privileged process, etc. Depending on the level (and how smart the agent is), certain type and amount of metadata can be collected.

I first thought about the agent approach since I was concerned that people may not have the time/resources to populate MDAS catalogs with appropriate metadata. However, the basic concept can be extended to glean lots of information by monitoring system activity. In fact, systems like Andrew Gross' intrusion analysis s/w and the network weather service may be collecting all sorts of information that may of interest to MDAS. Since the agents embody MDAS' efforts to include as many resources into MDAS as possible, we can think of this as the MDAS Outreach Program!

4.16 File I/O Interface to Archival Storage

This section describes an early implmentation of a set of MDAS API's which provide access transparency to applications accessing data sets stored in archival storage systems. This is achieved by providing a common file I/O interface to various archival storage systems. Thus, regardless of where a data set resides (on disk or in various archival storage systems), the interface to access the data set is the same. As explained below, these particular API's do not provide location transparency since location information needs to be provided as input to them. However, the intent in MDAS is to implement higher level API's which can query MDAS metadata to obtain the location information required by the lower level API's described here. The file I/O interface described here refers to a particular implementation of the interface at the San Diego Supercomputer Center (SDSC) which uses the Postgres95 database management system to provide an interface to the NSL UniTree and the IBM High Performance Storage System (HPSS) archival storage systems. NSL UniTree is a hierarchical archival storage system currently running in production mode at SDSC. The system is capable of storing almost unlimited amount of data. This system will be replaced by the HPSS system by the beginning of 1997.

4.16.1 The File I/O API

Currently, the file I/O API specification includes the following standard file I/O calls, *create, open, unlink, stat, read, write, seek, sync, close, stat*. The function input/output parameters for these functions are the same as those for the corresponding UNIX calls, except in the case of the first four functions, viz. *create, open, unlink, stat*. For these functions, two additional parameters are introduced which are described below.

New Parameters.

The file I/O interface introduces two new parameters, *access method* and *host address*, which are used to specify the location of a data set in a distributed system which supports replicated archives. These parameters are relevant only to functions that refer to a file by its name rather than by its file *handle*.

The "Access Method" parameter.

The access method parameter, *access_m*, is used to specify the access method for storing/retrieving data sets. The data type is `integer` and each integer value maps to a specific method of access. For example:

- 0 – local disk
- 1 – UniTree
- 2 – HPSS

The "Host Address" parameter.

The host address parameter, *host_addr*, specifies the address of the host system where the data resides. This provides support for remote sites in a distributed system.

Interface Functions.

Following is a list of functions that together form the file I/O interface to archival storage systems. The prefix "*e_*" is used to distinguish these calls from the standard UNIX file I/O calls.

Functions with filename as parameter.

1. *Create.*

```
int e_create(int access_m, text *host_addr, text *filename,
                    int mode)
```

2. *Open.*

```
int e_open(int access_m, char *host_addr, char *filename,
                    int flags, int mode)
```

3. *Unlink.*

```
int e_unlink(int access_m, char *host_addr, char *filename)
```

4. *Stat.*

```
int e_stat(int access_m, char *host_addr, char *filename,
                    struct stat *statbuf)
```

For example, the following call to the `e_create` function will create the file, *DOCT_archive_file* on the HPSS archival storage system running on host, *suraj.sdsc.edu*:

```
e_create(2, "suraj.sdsc.edu", "DOCT_archive_file", 0)
```

Functions with file handle as parameter.

1. *Close.*

```
int e_close(int fd_inx)
```

2. *Read.*

```
int e_read(int fd_inx, char *buf, int len)
```

3. *Write.*

```
int e_write(int fd_inx, char *buf, int len)
```

4. *Seek.*

```
int e_seek(int fd_inx, long offset, int whence)
```

5. *Sync.*

```
int e_sync(int fd_inx)
```

4.16.2 A Prototype Implementation

A prototype of the file I/O interface has been implemented at SDSC for the NSL UniTree and IBM HPSS systems. The Postgres95 database management system was used as the front-end to this archival storage system. The Postgres DBMS serves two purposes. First, it provides a means for storing metadata associated with the archival data sets that are stored in UniTree/HPSS. Using a DBMS allows one to query the metadata using the full power of standard database query language interfaces. Second, the Postgres system provides the means to implement the functions specified in the file I/O interface as internal Postgres functions. Thus, applications can store/retrieve archival data by directly interacting with the Postgres DBMS.

Implementation using the Postgres DBMS.

Each of the functions in the interface is implemented as an internal function in the Postgres *server*. Corresponding functions are also implemented on the client side, to support client/server access to the data sets. Accessing archived data sets requires making a connection to the Postgres server from the client using the Postgres CONNECT API. Subsequent calls from the client need to specify the corresponding "connection handle" to identify the established connection.

The client-side functions are the same as the corresponding functions on the server-side with the addition of one parameter, viz. the Postgres connection descriptor called, *PGconn*. This descriptor is returned by the call to the CONNECT API. For example, the client function prototype for *create* is follows:

```
int e_create(PGconn* conn, int access_m, char *host_addr,
                 char *filename, int mode)
```

Authorization.

The file interface must include checking to ensure that users have the neccesary authorization to execute the various functions on the specified data sets in the specified archival systems. Each function can verify this for the corresponding user. In the prototype implementation, authentication is done at two levels. User authentication is done during Postgres CONNECT processing to ensure users have the authorization to connect to the DBMS. Authorization at the file level is done during file open and create (*e_open* and *e_create*) processing to ensure users have the authorization to perform the specific I/O operation. In general, it is expected that security and authorization checking on data sets will be done by the *ticket* services provided by MDAS.

4.16.3 Alternative Implementations

In general, it is expected that a DBMS will be used to store all the metadata associated with an archival data set. However, there are several alternative implementations for providing access to the archival data set itself. These include storing the data set as a file under the control of a DBMS, as a large object within a DBMS, and simply as an external file (outside of a DBMS). These alternatives are further described in the following sections.

DBMS-based implementations.

The DBMS-based implementations can be divided into implementations that use an "external" file implementation versus those that employ the large object support provided by the DBMS itself.

External file implementation.

The prototype described in this report is an example of a DBMS-based, external file implementation. The DBMS implements internal functions corresponding to the functions specified in the file interface. These functions are made available to application programs as DBMS API's. Thus, application programs can directly call these functions via the API. The application program passes read/write buffers to the DBMS which, in turn, passes these buffers to the archival storage system.

Large Object Implementation.

Object-relational database systems have various mechanisms by which they are able to support the storage and manipulation of so-called *large objects* within a DBMS. In this implementation, the archival data sets are stored as regular large objects with the difference being that the DBMS is able to direct these objects to archival storage systems rather than storing them on local disk. The functions specified in the interface can be implemented specifically for large objects within a database. Alternatively, existing methods provided by the DBMS for handling large objects can be overloaded to handle archival data sets as well.

A Prototype based on Postgres Large Objects.

Postgres95 supports the storage and manipulation of large objects within the database. Thus, a prototype of the archival storage file interface was implemented by extending the existing Postgres large object implementation to allow for multiple storage/access methods. In Postgres, each large object is associated with an object id (Oid) which is assigned when the large object is created. The function prototypes on the DBMS server side are as follows:

```
Oid lo_creat(int access_m)

int lo_open(Oid object_id, int mode)

int lo_unlink(Oid object_id)

int lo_close(int fd)

int lo_read(int fd, char *buf, int len)

int lo_write(int fd, char *buf, int len)

int lo_seek(int fd, int offset, int whence)

Oid lo_import (char *filename, int access_m)

int lo_export (Oid object_id, char *filename)
```

Except for *lo_import* and *lo_export*, these functions are similar to the ones in the external file implementation and adhere to the same UNIX file I/O paradigm. *Lo_import* is used to import a UNIX file as a large object and *lo_export* is used to create a UNIX file from a large object. As before, the *access_m* parameter in the *lo_creat* and *lo_import* functions allows the client to choose the access method for the large object to be created.

As before, corresponding functions are implemented on the client side. They are the same as the server API's with the inclusion of the the database connection descriptor parameter in each case, e.g.

```
Oid lo_creat(PGconn* conn, int access_m)
```

Non-DBMS Implementation.

In a non-DBMS implementation, access to the archival data sets is provided by a service outside of the DBMS which supports the file I/O interface. Thus, the interface is implemented as a separate, stand-alone service. As mentioned before, a DBMS may still be used to store metadata related to the archival data sets. However, to access a data set, the application program interacts directly with the service.

Chapter 5

Important Findings and Conclusions

The primary challenges in the MDAS project are (a) the integration of data management systems with archival storage and (b) end-user solutions for the replacement of the (Unix) file paradigm with a higher-order interface to data, methods, and resources. MDAS will develop robust prototype solutions to these challenges, but several general problems will remain that merit follow-on efforts. These include:

Intelligent Hierarchical Storage Systems : Current HSS technology is designed for either (a) atomic file input/output by many users or (b) general read, write, and seek operations by a few users. An internal intelligent queueing mechanism is desirable to scale general I/O capabilities to a large number of simultaneous user requests.

Software Development Environment Standards : The lack of standards in system software tools makes developing multi-platform software a tedious process. The further development and acceptance of POSIX standards for Unix will provide some relief in this area. The situation is particularly acute in high performance computing. In 1995, the NCO for HPCC sponsored a report by the System Software Tools Working Group (SSTWG) on desired standards in system software tools. Reference to this report in procurements is likely to have a major effect on vendor compliance.

Heterogeneous MPI : The MPI Forum is defining protocols which will enable the communication of data-type structures and file structures between third-party applications. Further, MPI is defining an interoperable storage description that will allow the same binary file to be read by different binary format computing platforms. These capabilities are restricted to applications running the same version (implementation) of MPI. Further, these communication and storage modes are optional to the user so that applications desiring higher-performance communication and storage protocols may have them on vendor-specific architecture. However, it is likely that no single implementation of the MPI standard will run on all platforms of interest to a particular site; i.e., there is still a need for interoperable *implementations* of MPI. This could be acheived by defining an optional interoperable communication mode in the MPI standard itself.

Advanced OO technology : Object-oriented (OO) software technologies greatly simplify the task of software engineering and hold great promise for software reuse. However, present-day OO compilers do not produce high-performance executables. Improvements to both OO languages and compilers would be of great benefit.

Dynamic Loading : A "just-in-time" compiled applet is essentially a dynamically loaded software module. Dynamic loading has existed in some Unix compiler technology for several years, but is not standard practice. It is particularly useful in data mining and analysis applications for compiled source code derived from symbolic mathematics and query languages. Dynamic unloading is equally desirable when a module is no longer needed.

Universal Resource Names : Scientific applications should be able to access data and cache it locally no matter where the data is originally located. This is equivalent to requiring a catalog or expert system with universal resource name (URN) capabilities.

Resource Discovery : Current O/S technologies do not provide adequate interfaces for resource discovery. For example, to "discover" that a particular DBMS is running on a remote platform, the user must perform manual work to find the port number and appropriate library interface. SNMP provides a partial solution. A general O/S independent mechanism for automated discovery is needed. Meeting site dependent security needs will be an important aspect.

Parallel I/O : Support for parallel I/O streams must be done within the context of emerging standards. This requires tracking the MPI2 IO effort which is examining issues related to message passing within a compute platform and I/O to external peripherals. Interoperability between MPI and non-MPI processes will require specialized software interfaces.

Distributed Computation Support : Data sets may be distributed to multiple platforms, for analysis by methods that are retrieved from a DBMS. Support for distribution of computation objects is needed.

Third-party Authentication : Methods and data sets need to validate their interoperation through an authentication mechanism that is independent of the local operating system.

Common Communication Layer : Many of the above problems could be solved by a standardized communication layer that addresses concerns across all the sectors of the computing community. At present, there are many protocols and software implementations available with limited capabilities from which general prototypes can be developed. The NEXUS system from Argonne National Laboratory is example.

Chapter 6

Implications For Further Research

The issues researched in MDAS are essential in enabling the "Distributed Object Computation Testbed" project which will build a complex document handling system on top of federated databases that access replicated archives. The integration of database, archival storage and application (Web in particular) technology promises to facilitate the manipulation of large data sets and large collections of data sets. One goal is to enable data analysis on terabyte-sized data sets retrieved from petabyte archives, at an access rate of 10 GB/sec. Current supercomputer technology supports a 1 GB/s access rate to 1 terabyte of disk. For a teraflops supercomputer with 10 TB of disk, data rates on the order of 10 GB/s will be feasible. This will require, however, support for parallel I/O streams, and support for striping data sets across multiple peripherals. Fortunately, the software technology to support third party transport of data sets across parallel I/O streams is being developed in the HPSS archival storage system. Data redistribution mechanisms for the parallel data streams are being standardized as part of the MPI-IO effort. The expectation is that the initial usage prototypes described above can be extended to support supercomputer applications that analyze arbitrarily large data sets.

Bibliography

[1] S.K. Chang, R. Korfhage, S. Levialdi, and T. Ichikawa. Ten Years of Visual Language Research. In *Proc. IEEE Symposium on Visual Languages*, page 196, 1994.

[2] F. Davis, W. Farrell, J. Gray, R. Mechoso, R. Moore, S. Sides, and M. Stonebraker. EOSDIS Alternative Architecture. Technical report, San Diego Supercomputer Center. 1/23/95, http://www.research.microsoft.com/research/barc/gray/eos_dis/.

[3] S. Fineberg. MPI-IO: A Parallel File I/O Interface for MPI. http://lovelace.nas.nasa.gov/MPI-IO/.

[4] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI.* MIT Press, 1994.

[5] S. Tuecke I. Foster. Enabling Technologies for Web-Based Ubiquitous Supercomputing. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing.* ftp://ftp.mcs.anl.gov/pub/nexus/reports/hpdc_java.ps.gz.

[6] John F. Karpovich. Support for Object Placement in Wide Area Heterogeneous Distributed Systems. Technical report, Univ. of Virginia CS-96-03. 1/96, ftp://ftp.cs.virginia.edu/pub/techreports/CS-96-03.ps.Z.

[7] Sandia National Laboratory. Sandia Joins R&D Effort To Bring Teraflop Supercomputing On-Line. http://www.cs.sandia.gov/teraflop.html.

[8] R. Moore. High Performance Data Assimilation, a White Paper. Technical report, San Diego Supercomputer Center, http://www.sdsc.edu/EnablingTech/InfoServers/HPDA.html.

[9] MPI Forum. MPI 2. http://www.mcs.anl.gov/mpi/mpi2/mpi2.html.

[10] Andreas Paepcke, Steve B. Cousins, Hector Garcia-Molina, Scott W. Hassan, Steven P. Ketchpel, Martin Rvscheisen, and Terry Winograd. Using Distributed Objects for Digital Library Interoperatbility. *IEEE Computer*, May 1996. http://www.computer.org/pubs/computer/dli/r50061/r50061.htm.

[11] Scalable I/O Initiative. Working Papers. http://www.ccsf.caltech.edu/SIO/SIOpubslist.html.

[12] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference.* MIT Press, 1995.